1. **Признаки сложных систем. Сложность, присущая ПС.**

1. Сложные системы часто являются иерархическими и состоят из взаимосвязанных подсистем, которые в свою очередь могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня.

2. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большей степени оставляется на усмотрение исследователя.

3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

5. Любая работающая сложная система является результатом развития работавшей более простой системы… Сложная система, спроектированная с «нуля», никогда не заработает. Следует начинать с работающей простой системы.

Любая сложная система основана на процессах, которые определяют существо этой системы и которые остаются неизменными во времени. Однако реализация этих процессов может постоянно видоизменяться, принимая различные формы. Сложная система не может иметь детальной формализации в силу своей сложности. Попытка составить детальное формальное описание такой системы приводит к слишком высоким расходам человеческих, временных и финансовых ресурсов. За то время пока создаётся детальное формальное описание, предметная область или наши знания о ней могут существенно измениться, что сделает часть описания непригодным для использования.

*Сделаем выводы о том, что сложная система:*

1. основана на ограниченном числе базовых процессов, определяющих существо системы;

2. не имеет детального формального описания;

3. обладает выраженной иерархичностью;

4. находится в постоянном развитии и, как следствие, не имеет завершённого состояния;

5. разложима на более простые составляющие, вплоть до элементарных;

6. определяется сложностью связей между её составляющими;

7. выполняет только функции управления по отношению к составляющим;

1. **Классический жизненный цикл ПС.**

Очень часто классический жизненный цикл называют каскадной или водопадной моделью, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе (рис. 1.1).

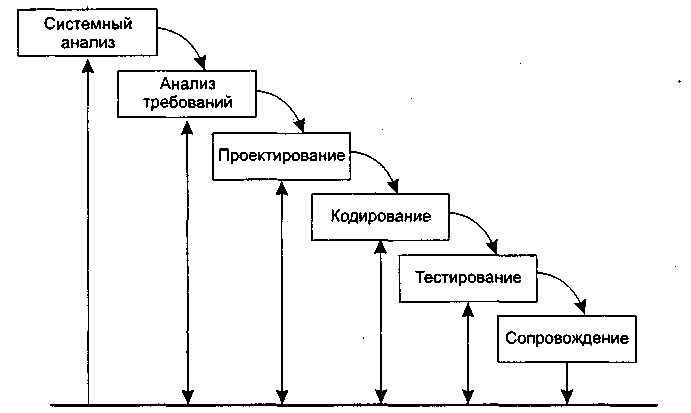
Охарактеризуем содержание основных этапов.

Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

***Системный анализ***задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному «элементу». Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

***Анализ требований***относится к программному элементу — программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.

Все определения документируются в *спецификации анализа.* Здесь же завершается решение задачи планирования проекта.



**Рис. 1.1.** Классический жизненный цикл разработки ПО

Проектирование состоит в создании представлений:

* архитектуры ПО;
* модульной структуры ПО;
* алгоритмической структуры ПО;
* структуры данных;
* входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в *спецификации анализа,* то есть в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

***Кодирование***состоит в переводе результатов проектирования в текст на языке программирования.

***Тестирование***— выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

***Сопровождение*** *—* это внесение изменений в эксплуатируемое ПО. Цели изменений:

* исправление ошибок;
* адаптация к изменениям внешней для ПО среды;
* усовершенствование ПО по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе, но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

***Достоинства классического жизненного цикла****:* дает план и временной график по всем этапам проекта, упорядочивает ход конструирования.

***Недостатки классического жизненного цикла****:*

1) реальные проекты часто требуют отклонения от стандартной последовательности шагов;

2) цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);

3) результаты проекта доступны заказчику только в конце работы.

1. **Итерационный и эволюционный жизненный цикл ПС.**

Общепринятая модель жизненного цикла является идеальной уже потому, что только очень простые задачи проходят все этапы без каких-либо итераций — возвратов на предыдущие шаги производственного процесса. При программировании, например, может обнаружиться, что реализация некоторой функции очень громоздка, неэффективна и вступает в противоречие с требуемой от системы производительностью. В этом случае необходимо перепроектирование, а может быть, и переделка спецификаций. При разработке больших нетрадиционных систем итеративность возникает регулярно на любом этапе жизненного цикла как из-за допущенных на предыдущих шагах ошибок и неточностей, так и из-за изменений внешних требований к условиям эксплуатации системы.

Существуют **3 стратегии** конструирования ПО:

* *однократный проход* (водопадная стратегия) — линейная последовательность этапов конструирования;
* *инкрементная стратегия.* В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система;
* *эволюционная стратегия.* Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

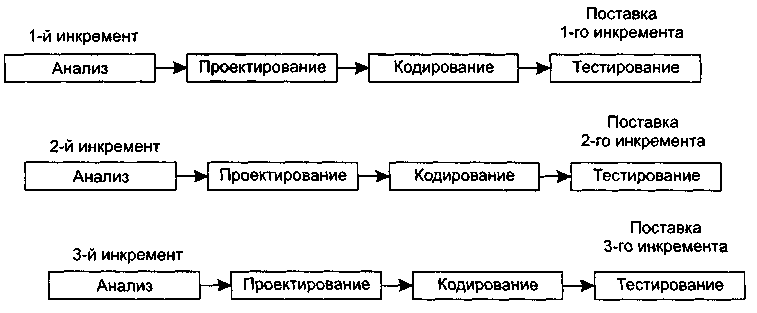
***Инкрементная модель***

Инкрементная модель является классическим примером инкрементной стратегии конструирования (рис. 1.4). Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными).

План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но, в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.



1. **Современные методы разработки ПП: ХР-процесс**

**Экстр. программирование**— облегченный (подвижный) процесс (или методология) разработки ПО (основан Кентом Беком в 1996). ХР-процесс ориентирован на группы малого и среднего размера, строящие программное обеспечение в условиях неопределенных или быстро изменяющихся требований. ХР-группу образуют до 10 сотрудников, которые размещаются в одном помещении. Динамизм обеспечивается с помощью четырех характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

**Основная идея ХР** — устранить высокую стоимость изменения. ХР-процесс должен быть высокодинамичным процессом. ХР-группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из очень коротких итераций. 4 базовые действия в ХР-цикле: кодирование, тестирование, выслушивание заказчика и проектирование.

Базис ХР образуют перечисленные ниже двенадцать методов.

1. Игра планирования (Planning game) — быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).

2. Частая смена версий (Small releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.

3. Метафора (Metaphor) — вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.

4. Простое проектирование (Simple design) — проектирование выполняется настолько просто, насколько это возможно в данный момент.

5. Тестирование (Testing) — непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций. «Тестируй, а затем кодируй» означает, что входным критерием для написания кода является «отказавший» тестовый вариант.

6. Реорганизация (Refactoring) — система реструктурируется, но ее поведение не изменяется; цель — устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.

7. Парное программирование (Pair programming) — весь код пишется двумя программистами, работающими на одном компьютере.

8. Коллективное владение кодом (Collective ownership) — любой разработчик может улучшать любой код системы в любое время.

9. Непрерывная интеграция (Continuous integration) — система интегрируется и строится много раз в день, по мере завершения каждой задачи. Непрерывное регрессионное тестирование, то есть повторение предыдущих тестов, гарантирует, что изменения требований не приведут к регрессу функциональности.

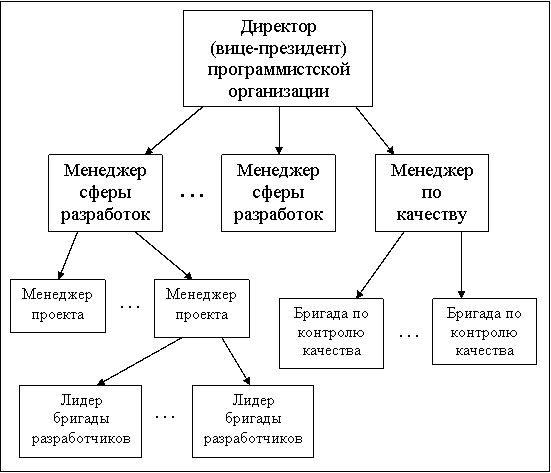
10. 40-часовая неделя (40-hour week) — как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.

11. Локальный заказчик (On-site customer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.

12. Стандарты кодирования (Coding standards) — должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы

1. **Структура управления разработкой программного средства.**

Разработка ПС обычно производится в организации, в которой одновременно могут вестись разработки ряда других программных средств. Для управления всеми этими программными проектами используется иерархическая структура управления.



***директор***программистской организации, (управляет всеми разработками ПС, решение о начале/прекращении проекта, обсуждение общих организ-х требований к проекту, решение возникающих проблем, требующих использование общих ресурсов программистской организации или изменения заказчиком общих требований)

***менеджер сферы разработок*** (управление разработками ПС определенного типа, формирование кол-ва исполнителей по проекту, осуществляет обобщение опыта разработок в своей сфере, накопление программных средств и документов для повторного использования.)

***менеджер проекта*** (назначается на каждый прогр. проект, осущ-ет планирование и составление расписаний работы бригад)

***бригада*** (большие проекты разбиваются в несколько относительно независимых подпроектов таким образом, чтобы каждый подпроект мог быть выполнен отдельной небольшой бригадой разработчиков)

* + - * обычные бригады (лидер бригады руководит работой младших программистов. Ошибки одного программиста могут препятствовать работе других программистов.)
      * неформальные демократические бригады (проект обсуждается всеми членами бригады, задания распределяются в зависимости от способностей и опыта )
      * бригады ведущего программиста (Ядро бригады - ведущий программист , его дублер, администратор базы данных разработки).

***менеджер по качеству –***управление обеспечением качества. Каждый этап разр-ки заканчивается смотром по контролю качества - функцией управления разработкой, связан с оценкой того, насколько результаты этой работы согласуются с декларированными требованиями относительно качества ПС.

Существенную роль в управлении качеством ПС играют программные (софтверные) стандарты – они фиксируют удачный опыт высоко квалифицированных специалистов по разработке. Различают два вида таких стандартов:

* + - * стандарты ПС (программного продукта),
      * стандарты процесса создания и использования ПС.

***Стандарты ПС***определяют некоторые свойства, которыми должны обладать программы или документы ПС. (стандарты на языки программирования, на состав документации, на структуру различных документов, на различные форматы и другие).

***Стандарты процесса создания и использования ПС*** определяют, как должен проводится этот процесс, т.е. подход к разработке ПС, структуру жизненного цикла ПС и его технологические процессы.

***Бригада по контролю качества*** состоит из ассистентов по качеству ПС. Она проводит смотр ПС с целью поиска возникающих проблем в процессе его разработки, с учетом требований, сформулированных в спецификации качества ПС. Результат – замечания разработчикам (устные/письменные)

1. **Метрики и методы оценки объема работ.**
2. **Внешнее описание программного средства.**

Внешнее описание ПС играет роль точной постановки задачи, решение которой должно обеспечить разрабатываемое ПС. Более того, оно должно содержать всю информацию, которую необходимо знать пользователю для применения ПС. Оно является исходным документом для трех параллельно протекающих процессов: разработки текстов (конструированию и кодированию) программ, входящих в ПС, разработки документации по применению ПС и разработки существенной части комплекта тестов для тестирования ПС. Ошибки и неточности во внешнем описании, в конечном счете, трансформируются в ошибки самой ПС и обходятся особенно дорого, во-первых, потому, что они делаются на самом раннем этапе разработки ПС, и, во-вторых, потому, что они распространяются на три параллельных процесса. Это требует принятия особенно серьезных мер по их предупреждению.

Исходным документом для разработки внешнего описания ПС являются ***определение требований*** к ПС. Но так как через этот документ передается от заказчика (пользователя) к разработчику основная информация относительно требуемого ПС, то формирование этого документа представляет собой довольно длительный и трудный итерационный процесс взаимодействия между заказчиком и разработчиком, с которого и начинается этап разработки требований к ПС.

В определении внешнего описания легко бросаются в глаза две самостоятельные его части. Описание поведения ПС определяет функции, которые должна выполнять ПС, и потому его называют ***функциональной спецификацией*** ПС. Функциональная спецификация определяет допустимые фрагменты программ, реализующих декларированные функции. Требования к качеству ПС должны быть сформулированы так, чтобы разработчику были ясны цели, которые он должен стремиться достигнуть при разработке этого ПС. Эту часть внешнего описания будем называть *спецификацией качества* ПС (в литературе ее часто называют *нефункциональной спецификацией*, но она, как правило, включает и требования к технологическим процессам). Она, в отличие от функциональной спецификации, представляется в неформализованном виде и играет роль тех ориентиров, которые в значительной степени определяют выбор подходящих альтернатив при реализации функций ПС, а также определяет стиль всех документов и программ требуемого ПС. Тем самым, спецификация качества играет решающую роль в обеспечении требуемого качества ПС.

Обычно разработка спецификации качества предшествует разработке функциональной спецификации ПС, так как некоторые требования к качеству ПС могут предопределять включение в функциональную спецификацию специальных функций, например, функции защиты от несанкционированного доступа к некоторым объектам информационной среды. Таким образом, структуру внешнего описания ПС можно выразить формулой:

Внешнее описание ПС = определение требований + спецификация качества ПС  + функциональная спецификация ПС

Таким образом, внешнее описание определяет, что должно делать ПС и какими внешними свойствами оно должно обладать. Оно должно достаточно точно и полно определять задачи, которые должны решить разработчики требуемого ПС. В то же время оно должно быть понято представителем пользователем - на его основании заказчиком достаточно часто принимается окончательное решение на заключение договора на разработку ПС. Внешнее описание играет большую роль в обеспечении требуемого качества ПС, так как спецификация качества ставит для разработчиков ПС конкретные ориентиры, управляющие выбором приемлемых решений при реализации специфицированных функций.

1. **Основные классы архитектур программных средств.**

**Архитектура ПС** - это представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы.

Разработка архитектуры является первым этапом борьбы со сложностью ПС, на котором реализуется принцип выделения относительно независимых компонент.

**Основные задачи разработки архитектуры ПС:**

1)выделение программных подсистем и отображение на них внешних функций (заданных во внешнем описании) ПС;

2)определение способов взаимодействия между выделенными программными подсистемами.

**Основные классы архитектур программных средств:**

1)цельная программа;

2)комплекс автономно выполняемых программ;

3)слоистая программная система;

4)коллектив параллельно выполняемых программ.

**Цельная программа** представляет вырожденный случай архитектуры ПС: в состав ПС входит только одна программа. Такую архитектуру выбирают обычно в том случае, когда ПС должно выполнять одну какую-либо ярко выраженную функцию и ее реализация не представляется слишком сложной. Естественно, что такая архитектура не требует какого-либо описания (кроме фиксации класса архитектуры), так как отображение внешних функций на эту программу тривиально, а определять способ взаимодействия не требуется (в силу отсутствия какого-либо внешнего взаимодействия программы, кроме как взаимодействия ее с пользователем, а последнее описывается в документации по применению ПС).

**Комплекс автономно выполняемых программ** состоит из набора программ, такого, что:

1)любая из этих программ может быть активизирована (запущена) пользователем;

2)при выполнении активизированной программы другие программы этого набора не могут быть активизированы до тех пор, пока не закончит выполнение активизированная программа;

3)все программы этого набора применятся к одной и той же информационной среде.

Таким образом, программы этого набора по управлению никак не взаимодействуют - взаимодействие между ними осуществляется только через общую информационную среду.

**Слоистая программная система** состоит из некоторой упорядоченной совокупности программных подсистем, называемых слоями, такой, что:

1)на каждом слое ничего не известно о свойствах (и даже существовании) последующих (более высоких) слоев;

2)каждый слой может взаимодействовать по управлению (обращаться к компонентам) с непосредственно предшествующим (более низким) слоем через заранее определенный интерфейс, ничего не зная о внутреннем строении всех предшествующих слоев;

3)каждый слой располагает определенными ресурсами, которые он либо скрывает от других слоев, либо предоставляет непосредственно последующему слою (через указанный интерфейс) некоторые их абстракции.

**Коллектив параллельно действующих программ** представляет собой набор программ, способных взаимодействовать между собой, находясь одновременно в стадии выполнения. Это означает, что такие программы, во-первых, вызваны в оперативную память, активизированы и могут попеременно разделять по времени один или несколько центральных процессоров, а во-вторых, осуществлять между собой динамические (в процессе выполнения) взаимодействия, на базе которых производиться их синхронизация. Обычно взаимодействие между такими процессами производится путем передачи друг другу некоторых сообщений.

1. **Понятие программного модуля и его основные характеристики.**

Приступая к разработке каждой программы ПС, следует иметь ввиду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями. Программный модуль - это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю.

**Основные характеристики программного модуля:**

* хороший модуль снаружи проще, чем внутри;
* хороший модуль проще использовать, чем построить.

Размер модуля измеряется числом содержащихся в нем операторов (строк). Модуль не должен быть слишком маленьким или слишком большим.

**Прочность модуля** - это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести

**Функционально прочный модуль** - это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули.

**Информационно прочный модуль** - это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему.

**Сцепление модуля** - это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей.

**Рутинность модуля** - это его независимость от предыстории обращений к нему.

1. **Методы разработки структуры программ.**

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода: **метод восходящей разработки и метод нисходящей разработки.**

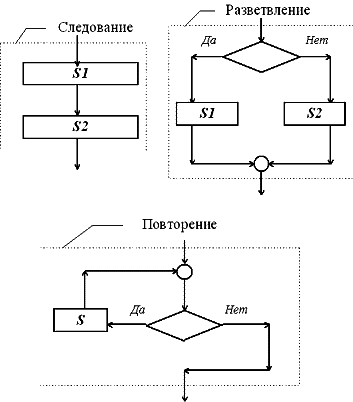
**Метод восходящей разработки** заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование.

**Метод *нисходящей разработки*** заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При этом первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при естественном состоянии информационной среды, при котором начинает выполняться эта программа. При этом те модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми заглушками). Каждый *имитатор* модуля представляется весьма простым программным фрагментом, который, в основном, сигнализирует о самом факте обращения к имитируемому модулю, производит необходимую для правильной работы программы обработку значений его входных параметров (иногда с их распечаткой) и выдает, если это необходимо, заранее запасенный подходящий результат.

1. **Структурное программирование и пошаговая детализация. Псевдокод.**

***Структурное программирование.***

Построение программы как композиции из нескольких типов управляющих конструкций (структур), которые позволяют сильно повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали ***структурным***.



Основными конструкциями структурного программирования являются: следование, разветвление и повторение (см. Рис.5). Компонентами этих конструкций являются обобщенные операторы (узлы обработки) S, S1, S2 и условие (предикат) P. В качестве обобщенного оператора может быть либо простой оператор используемого языка программирования (операторы присваивания, ввода, вывода, обращения к процедуре), либо фрагмент программы, являющийся композицией основных управляющих конструкций структурного программирования. Существенно, что каждая из этих конструкций имеет по управлению только один вход и один выход. Тем самым, и обобщенный оператор имеет только один вход и один выход.

Весьма важно также, что эти конструкции являются уже математическими объектами (что, по существу, и объясняет причину успеха структурного программирования). Доказано, что для каждой неструктурированной программы можно построить функционально эквивалентную (т.е. решающую ту же задачу) структурированную программу. Для структурированных программ можно математически доказывать некоторые свойства, что позволяет обнаруживать в программе некоторые ошибки.

***Пошаговая детализация и понятие о псевдокоде.***

В качестве основного метода построения текста модуля современная технология программирования рекомендует *пошаговую детализацию.* Сущность этого метода заключается в разбиении процесса разработки текста модуля на ряд шагов.

На первом шаге описывается общая схема работы модуля в обозримой линейной текстовой форме (т.е. с использованием очень крупных понятий), причем это описание не является полностью формализованным и ориентировано на восприятие его человеком. На каждом следующем шаге производится уточнение и детализация одного из понятий (будем называть его *уточняемым*), в каком либо описании, разработанном на одном из предыдущих шагов. В результате такого шага создается описание выбранного уточняемого понятия либо в терминах базового языка программирования (т.е. выбранного для представления модуля), либо в такой же форме, что и на первом шаге с использованием новых уточняемых понятий. Этот процесс завершается, когда все уточняемые понятия будут *уточнения* (т.е. в конечном счете будут выражены на базовом языке программирования). Последним шагом является получение текста модуля на базовом языке программирования путем замены всех вхождений уточняемых понятий заданными их описаниями и выражение всех вхождений конструкций структурного программирования средствами этого языка программирования.

Пошаговая детализация связана с использованием частично формализованного языка для представления указанных описаний, который получил название *псевдокода*. Этот язык позволяет использовать все конструкции структурного программирования, которые оформляются формализовано, вместе с неформальными фрагментами на естественном языке для представления обобщенных операторов и условий.

1. **Тестирование и отладка программных средств.**

***Отладка* ПС** − это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ. ***Тестирование* ПС** − это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется *тестовым* или просто *тестом*. Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование.

*1)Автономная отладка программного средства:*

При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть − модулями, управляющими отладкой (отладочными модулями, см. ниже). Таким образом, при автономной отладке тестируется всегда некоторая программа (тестируемая программа), построенная специально для тестирования отлаживаемого модуля.

*2)Комплексная отладка программного средства:*

**Тестирование архитектуры ПС***.* Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС.

**Тестирование внешних функций**. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС

**Тестирование качества ПС***.* Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС.

**Тестирование документации по применению ПС***.* Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС.

**Тестирование определения требований к ПС.** Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему.

1. **Объектно-ориентированный подход к разработке ПС.**

Объектно-ориентированное представление ПС основывается на принципах абстрагирования, инкапсуляции, модульности и иерархической организации.

***Абстрагирование*** – это отвлечение от несущественных характеристик конкретных объектов, выделение только существенных характеристик. (часы – время без учета формы , цвета.. )

***Инкапсуляция*** является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от ее реализации.

Инкапсуляция и абстракция — взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение.

Модульность определяет способность системы подвергаться декомпозиции на ряд сильно связанных и слабо сцепленных модулей. Общая цель декомпозиции на модули: уменьшение сроков разработки и стоимости ПС за счет выделения модулей, которые проектируются и изменяются независимо.

**Иерархическая организация** — формирование из абстракций иерархической структуры. Определением иерархии в проекте упрощаются понимание проблем заказчика и их реализация — сложная система становится обозримой человеком.

Двумя важными инструментами иерархической организации в объектно-ориентированных системах являются:

* структура из классов *(«is a»*-иерархия);
* структура из объектов *(«part of»*-иерархия).

**объектно-ориентированный подход помогает справиться с такими сложными проблемами, как**

1)уменьшение сложности программного обеспечения;

2)повышение надежности программного обеспечения;

3)обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;

4)обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. Объектно-ориентированный подход является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

1. **Унифицированный язык моделирования UML.**

**UML** — стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML — это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal) и даже в таблицы для реляционной БД.

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

**Предметы** — это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

**Язык UML предназначен для решения следующих задач:**

1)Предоставить в распоряжение пользователей легко воспринимаемый и выразительный язык визуального моделирования, специально предназначенный для разработки и документирования моделей сложных систем самого различного целевого назначения.

2)Снабдить исходные понятия языка UML возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области.

3)Описание языка UML должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем.

4)Описание языка UML должно включать в себя семантический базис для понимания общих особенностей объектно-ориентированного анализа и проектирования.

5)Поощрять развитие рынка объектных инструментальных средств.

6)Способствовать распространению объектных технологий и соответствующих понятий объектно-ориентированного анализа и проектирования.

7)Интегрировать в себя новейшие и наилучшие достижения практики объектно-ориентированного анализа и проектирования.

**Общая структура языка UML:**

С самой общей точки зрения описание языка UML состоит из двух взаимодействующих частей, таких как:

1)Семантика языка UML. Представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML.

2)Нотация языка UML. Представляет собой графическую нотацию для визуального представления семантики языка UML.

В UML имеются четыре разновидности предметов:

***Структурные предметы***являются существительными в UML-моделях. Представляют статические части модели — понятийные или физические элементы. Перечислим восемь разновидностей структурных предметов.

1. ***Класс***— описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов.
2. ***Интерфейс*** *—* набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне.
3. ***Кооперация***(сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов.
4. ***Актер*** *—* набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой. Каждая роль требует от системы определенного поведения.
5. ***Элемент Use Case***(Прецедент) — описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат.
6. ***Активный класс*** *—* класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность
7. ***Компонент*** *—* физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов
8. ***Узел*** *—* физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки

***Предметы поведения*** *—* динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

1. ***Взаимодействие***— поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели.
2. ***Конечный автомат*** *—* поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события. С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход).

***Группирующие предметы***— организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета — ***пакет*** *—* общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет существует только в период разработки.

***Поясняющие предметы*** *—* разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета — ***примечание*** *—* символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов

В UML имеются четыре разновидности отношений:

1. ***Зависимость***— семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета).
2. ***Ассоциация*** *—* структурное отношение, которое описывает набор связей, являющихся соединением между объектами.
3. ***Обобщение***— отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя.
4. ***Реализация*** *—* семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации).

***Диаграмма***— графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). UML включает девять видов диаграмм:

1) диаграммы классов;

2) диаграммы объектов;