**ЭТАПЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

**1. Этапы разработки программного обеспечения.**

1) анализ требований, предъявляемых к системе;

2) определение спецификаций;

3) проектирование;

4) кодирование;

5) тестирование:

а) автономное;

б) комплексное;

6) эксплуатация и сопровождение.

**2. Анализ требований, предъявляемых к системе.**

На этом этапе формулируются *целевое* назначение и основные *свойства* разрабатываемой программной системы.

В том случае, когда разработка программного обеспечения является самоцелью, обычно использу-

ются методы составления исходных описаний. Одним из самых эффективных методов исходных описаний является метод структурного анализа, сущность которого сводится к декомпозиции исходного объекта на его составные части

базовые требования:

−время обработки (работы) программы;

−стоимость обработки;

−вероятность ошибки;

−реакция на непредсказуемые действия оператора (за-

щита от дурака и др.).

Особое внимание следует уделять пространственно-временным ограничениям и средствам системы, которые в будущем могут претерпеть изменения

К важнейшим требованиям относятся ресурсные требования и затраты на реализацию системы.

Фактически, анализ требований завершается составлением развернутого технического задания на систему, которое в терминологии классического САПР называется аван-проектом.

**3. Жизненный цикл программного обеспечения. (в чем разница с п.1?)**

Для управления ходом разработки больших программных систем выделяются *шесть* этапов, составляющих цикл разработки (цикл жизни) программного обеспечения:

1) анализ требований, предъявляемых к системе;

2) определение спецификаций;

3) проектирование;

4) кодирование;

5) тестирование:

а) автономное;

б) комплексное;

6) эксплуатация и сопровождение.

**4. Функциональные спецификации. Определение спецификаций.**

На этапе определения спецификаций осуществляется точное *описание функций*, реализуемых ЭВМ, а также задаются *структура* входных и выходных данных, методы и средства их размещения. Определяются алгоритмы обработки данных.

В спецификациях должны быть представлены *данные* для тестирования элементов системы и системы в целом.

В общем случае, спецификации определяют те функции, которые должна выполнять система, не указывая, каким образом это достигается.

**5. Проектирование. Кодирование.**

Проектирование -

На стадии проектирования разрабатываются *алгоритмы*, задаваемые спецификациями, и формируется общая *структура* вычислительной системы. При этом система разбивается (при

необходимости) на составные части таким образом, чтобы ответственность за реализацию каждой составной части можно было бы возложить на одного разработчика (или группу исполнителей).

Кодирование -

Данный этап является наиболее простым, а его реализация существенно облегчается при использовании алгоритмических языков высокого уровня. Кодирование — это этап разработки программного обеспечения, доставляющий наименьшее беспокойство разработчику.

**6. Тестирование: программное, системное, оценочное и сравнительное.**

Тестирование подразумевает три стадии:

−автономное;

−комплексное и

−системное.

При *автономном* тестировании модуль проверяется с помощью данных, подготовленных программистом.

В процессе *комплексного* тестирования проводится совместная проверка групп программных компонент.

*Системное* (или оценочное) тестирование — это завершающая стадия проверки системы, т.е. проверка системы в целом с помощью независимых тестов.

Для случая, когда сравниваются характеристики нескольких систем (имеется альтернативная разработка), такая процедура известна как *сравнительное* тестирование.

**7. Сбой системы, выброс, ошибка. Испытания. Верификация системы.**

*Испытание* системы осуществляется посредством тестирования. Цель такой проверки — показать, что система функционирует в соответствии с разработанными на нее спецификациями.

*Верификация* заключается в выполнении доказательств, что программа удовлетворяет своим спецификациям.

Общий процесс создания правильных программ с помощью процедур испытания и верификации называется *аттестацией*.

*Сбой* системы — это явление, связанное с нарушением системой установленных на нее спецификаций.

Данные, при обработке которых правильными алгоритмами системы происходит сбой, называются *выбросом*.

*Ошибка* — это алгоритмический дефект, который создает выброс (программная ошибка).

**8. Правильность и надежность программ.**

*Правильная* программа — это та, что удовлетворяет своим спецификациям.

Что касается *надежной* программы, то она не обязательно является правильной, но выдает приемлемый результат даже в том случае, когда входные данные либо условия ее использования не удовлетворяют принятым допущениям.

Система является *правильной*, если в системе нет ошибок, а ее внутренние данные не содержат выбросов.

Система называется *надежной*, если, несмотря на сбои, она продолжает удовлетворительно функционировать.

**9. Эксплуатация и сопровождение. Периоды обновления.**

1) анализ требований;

2) определение спецификаций;

3) проектирование;

4) кодирование;

5) автономное тестирование;

6) комплексное тестирование;

7) сопровождение.

Чтобы исключить лавинообразное нарастание версий, системы обычно корректируются в определенные промежутки времени, называемые *периодами обновления*.

**МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КАК НАУЧНАЯ ДИСЦИПЛИНА**

1. **Методы разработки программного обеспечения как научная дисциплина.**

1) разработка методов управления сложными системами;

2) повышение надежности и правильности программного обеспечения;

3) развитие методов более точного прогнозирования затрат на создание программного обеспечения.

Методы *управления* разработкой имеют отношение к эффективной организации работы исполнителей.

Методы *проведения* разработки охватывают технические приемы работы программистов, способствующие повышению производительности их труда.

1. **Организация интерфейса между модулями, написанными разными программистами.**

Обычно наибольшие трудности возникают при построении интерфейса между модулями, написанными различными программистами. Поскольку количество таких интерфейсов при

*N* исполнителях соответствует *N*(*N*–1)/2 и возрастает пропорционально квадрату числа исполнителей, проблема становится довольно сложной при разработке проекта группой из 4-х и более человек.

1. **Выполнение проекта. Бригада главного программиста.**

В крупных организациях (фирма IBM) создается бригада *главного* программиста. При создании бригады исходят из того, что программисты имеют различные уровни квалификации. Организация бригады главного программиста является одним из путей уменьшения количества коммуникаций

1. **Методика оценки затрат. Методика инженерно-технической оценки затрат.**

Одним из важнейших аспектов процесса разработки программного обеспечения является оценка необходимых ресурсов или требуемых затрат.

*Шаг 1*. Формируются общие требования к системе исходя из существующего технического задания. Для более точного определения требуемых ресурсов проводится анализ требований.

*Шаг 2*. Собирается аналогичная информация, например данные о подобных системах.

*Шаг 3*. Отбираются основные релевантные данные.

*Шаг 4*. Проводится предварительная оценка.

*Шаг 4а*. Сравнение проектируемой системы с подобными уже разработанными системами.

*Шаг 4б*. Разбиение системы на части и сравнение каждой из этих частей с подобными ей частями других систем.

*Шаг 4в*. Планирование работ и оценка затрат на каждый месяц.

*Шаг 4г*. Разработка соглашений, которые могут быть использованы при работе.

*Шаг 5*. Проводится окончательная оценка системы.

1. **Методика экспертных оценок. Метод алгоритмического анализа. Пошаговый анализ. Закон Паркинсона.**

*Метод экспертных оценок.* Оценка производится исходя из личного опыта квалифицированного проектировщика (эксперта).

*Метод алгоритмического анализа.* При данном методе оценки затрат используется некоторый алгоритм. Оценка является объективной и повторяемой.

*Пошаговый анализ.* Задаются спецификации на основе пошагового анализа по нисходящему либо восходящему принципу, так что каждая определенная таким образом задача оценивается отдельно.

*Закон Паркинсона.* Во многих случаях для выполнения некоторой работы (задачи) затрачивается то время, которое отведено для нее, независимо от того, является ли выполнение этой работы необходимым.

1. **Затраты на завершение разработки.**

*Затраты на завершение разработки.* В некоторых случаях стоимость системы, оговариваемая при заключении договора, намеренно занижается разработчиком системы в надежде, что в последующем ее можно будет существенно увеличить за счет изменения спецификаций.

Заказчику следует проявлять осмотрительность в отношении предложений, резко отличающихся от других. Более того, он должен тщательно проводить анализ системных требований, чтобы избежать необходимости изменения спецификаций.

В общем случае точную оценку затрат можно дать лишь на основе опыта.

1. **Оценка длительности разработки на основе распределения Рэлея.**

В настоящее время некоторые результаты теории надежности аппаратных средств ЭВМ начинают использовать для оценки сроков и затрат при разработке программного обеспечения.

Поскольку 60 % затрат производится на этапе сопровождения, нет ничего удивительного, что максимум кривой близок к моменту создания системы, т.е. к тому моменту, когда традиционно считается, что работы завершены.

Кривая Рэлея имеет два параметра — *K* и *a*. В начале работы *K* можно оценить, используя величину планируемых затрат, а *a* можно определить, исходя из состава исполнителей. Дату за-

вершения работ определяют по достижению максимума расходов (максимум кривой Рэлея).

**8. Контрольные точки. Средства РАЗобработки. Надежность. Концептуальная целостность.**

*Контрольные точки* указывают на моменты завершенияработ; они позволяют судить о состоянии разработки системы. Контрольные точки планируются руководителем проекта с целью осуществления контроля за разработкой.

Существует большое количество «стандартных» контрольных точек:

−выпуск функциональных спецификаций;

−завершение проектирования отдельных модулей;

−компилирование модулей без ошибок;

−успешное проведение тестирования модуля и т.д.

Средства разработки:

Компиляторы и отладочные средства

Системы управления базами данных (**СУБД)**,

таблицы перекрестных ссылок, атрибутивные листинги, таблицы распределения памяти.

язык определения задач и анализатор определения задач (**PSL/PSA**)

система **RSL** (язык определения требований), предназначенная для определения требований и интерфейсов посредством системы управления данными.

Одним из основных параметров **надежности** разрабатываемой программной системы является *концептуальная целостность*, т.е. единообразие стиля и простота структуры.

Организация бригады главного программиста также способствует концептуальной целостности системы, т.к. основная работа по проектированию в этом случае выполняется главным про-граммистом.

Для контроля правильности используется так называемый *контрольный анализ*.

Цель контрольного анализа - обнаружение ошибок, а не их исправление.

1. **Верификация и испытания. Дамп. Трассировка. Анализ графов программ.**

*Верификация и испытания*. Верификация и испытания занимают почти половину времени, отведенного на создание системы.

Для тестирования программ используются *генераторы* тестовых данных. Проверка определенных условий в заданныхточках осуществляется с помощью *согласующих компиляторов*.

В качестве примера средства, предназначенного для отработки этапов определения спецификаций и проектирования, можно назвать систему **PSL/PSA**.

*Дамп* — это распечатка содержимого памяти.

*Трассировка* — это анализ значения данных переменных после каждого выполнения оператора.

*Анализаторы* графов программы способны выявить ситуацию, когда либо происходит обращение к неинициированной переменной, либо после присвоения значения переменной к ней не обращаются.

1. **«Уровни правильности» программ**.

Правильная программа:

1) не содержит синтаксических ошибок;

2) не имеет ошибок, допущенных в процессе компилирования, либо сбоев в процессе выполнения;

3) для некоторых наборов тестовых данных обеспечивает получение правильного результата;

4) для типичных наборов тестовых данных обеспечивает получение правильного результата;

5) для усложненных наборов тестовых данных обеспечивает получение правильного результата;

6) для всех возможных наборов данных, удовлетворяющих спецификации задачи, обеспечивает получение правильного результата;

7) для всех возможных наборов данных и всех вероятных условий ошибочного входа обеспечивает получение правильного результата;

8) для всех возможных наборов данных обеспечивает получение правильного результата.

Обычно для программных комплексов достаточно уровня правильности **6**.

Каждая программа правильна лишь по отношению к некоторому набору спецификаций

1. **Методы программирования. Эффективность программ.**

Эти методы наиболее развиты, т.к. они относятся к области профессиональной деятельности программистов: кодирование, тестирование, отладка и др. Средства их проведения (решения) известны давно и динамически развиваются. Довольно хорошо формализованы методы определения спецификаций. Здесь уже имеется эффективная методология, хотя не все технические проблемы преодолены.

Кроме традиционных хорошо известных методов разработки (проектирования) программного обеспечения, в настоящее время все шире применяются методы, ориентированные на автоматизированную разработку.

* Язык определения задач и анализатор задач (PSL/PSA/PDL)
* Система структурного анализа и проектирования SADT
* Система SREM
* Методика Джексона

1. **Определение спецификаций. Язык определения задач и анализатор определения задач (PSL/PSA**).

Среди таких методов следует отметить разработку Мичиганского университета **ISDOS:**

1. язык описания задач **PSL**, предназначенный для отображения функциональных требований и требований к ресурсам.
2. анализатор определения задач **PSA**, представляющий собой процессор, с помощью которого осуществляется испытание предложений, написанных на языке **PSL**.
3. язык проектирования программ **PDL.**

Язык **PSL** имеет простой синтаксис и ориентирован на использование ключевых слов. Основными операторами этого языка являются:

**PROCESS <имя>** <**имя**> определяется как новый процесс;

**DESCRIPTION <текст>** <**текст**> представляет описание функции, реализуемой процессом, средствами английского языка;

**SUBPARTS ARE <имя>** процесс <**имя**> связан с текущим процессом и расположен ниже его на дереве иерархии;

**PART OF <имя>** процесс <**имя**> вызывает текущий процесс;

**DERIVE <файл>** файл <**файл**> является выходным для текущего процесса;

**USING <файл>** файл <**файл**> является входным для текущего процесса;

**PROCEDURE <текст>** <**текст**> представляет описание на языке **PDL** алгоритма, реализуемого процессом.

*Пример*: если процесс **Y** содержит предложение **PART OF X**;

то процесс **X** должен включать предложение **SUBPARTS ARE Y**;.

Система **PSL/PSA** обладает следующими характеристиками:

1) позволяет пользователю получить формализованное представление проблемы;

2) осуществляет документирование системных требований в единообразной форме;

3) способствует выявлению условий возникновения некоторых видов ошибок, обусловленных, в первую очередь, неполнотой информации и нарушением последовательности ее ввода.

Язык **PDL** включает две категории структур:

- Имеется внешний синтаксис, построенный на основных типах операторов (**if-then-else**, **sequence** и др.). Внешний синтаксис предназначен для соединения отдельных компонент в

Программы

- внутренний синтаксис, соответствующий конкретному применению. Внутренний синтаксис выражается предложениями на естественном языке, которые раскрываются последовательно, пока алгоритм не окажется описанным компонентами внешнего синтаксиса.

1. **Система структурного проектирования SADT.**

**SADT** — это аббревиатура марки фирмы Software Technology, представляет собой ручную графическую систему, предназначенную для проектирования больших программных

комплексов.

Графический язык системы **SADT** — это иерархический структурированный набор диаграмм, причем каждый блок диаграммы раскрывается более детально с помощью другой диаграммы.

При использовании **SADT** каждый разработчик наделен строго определенными полномочиями.

Авторы*. -* Разработчики, занятые изучением требований и ограничений системы и их описаний с помощью системы **SADT**.

Комментаторы. - Обычно это проектировщики, анализирующие работу своих коллег (авторов) и подготавливающие замечания по ней.

Читатели. - Лица, занятые анализом проектов, разрабатываемых другими специалистами, но не обязанные их комментировать.

Технический комитет. - Группа опытных технических специалистов, анализирующих проект на высших уровнях его описания.

Библиотекарь проекта. - Лицо, отвечающее за ведение файлов проекта.

Руководитель проекта. - Лицо, несущее основную ответственность за техническую разработку проекта.

Главный аналитик. - Основной консультант по использованию **SADT**. Он хорошо понимает особенности использования системы **SADT** и выдает рекомендации по ее применению.

Инструктор. - Лицо, обучающее исследователей пользованию **SADT**.

- Основным и главным недостатком этой системы является тот факт, что она не автоматизирована.

1. **Структурное проектирование. Методика Джексона.**

Разработанная М. Джексоном методика включает нисходящее проектирование, структурное программирование и структурный контрольный анализ.

В соответствии с этой методикой строятся иерархически структурированные программы,

имеющие четыре компонента, подобные структурам управления в структурном программировании.

Элемент. - Функция, которая не может быть разбита на более простые функции.

Последовательность. - Ряд функций, реализуемых последовательно и однократно.

Выборка. - Одна из возможных последовательностей.

Итерация. Функция, выполняемая заданное число раз, включая нулевое.

Базовая идея метода состоит в том, что структура системы должна быть идентична структуре используемых данных. Следовательно, древовидная схема организации системы долж-

на отражать структуру данных: в противном случае проект будет неправильным.

**15. Стратегия объединения различных методов проектиро-**

**вания.**

Существует ряд стратегий объединения различных методов в единую методологию. Наиболее эффективную стратегию создал Боэм, который сформулировал семь принципов разработки программного обеспечения:

1. *управление разработкой на основе последовательной реализации отдельных этапов жизненного цикла программного обеспечения.*
2. *выполнение испытаний.* На каждом шаге совершенствования модуля осуществляется его аттестация;
3. *осуществление строгого контроля над ходом разработки.*
4. *использование всего диапазона средств структурного программирования.*
5. *строгий учет. Контрольные точки, системный журнал*
6. *использование немногочисленного штата, укомплектованного квалифицированными работниками.*
7. *высокий уровень.* Необходимо использовать имеющиеся достижения в области технологии и разработки программного обеспечения, не забывая при этом о надежности и возможности модификации программ.

**МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

1. **Язык проектирования программ PDL. Операторы выбора. Операторы цикла. Операторы описания данных. Операторы ввода/вывода и вызова процедур. Оператор leave. Предложения на естественном языке.**

Язык проектирования обычно состоит из двух частей:

−заданного набора операторов, построенных по образцу языков программирования;

−общего, обычно неопределяемого синтаксиса, пригодного для описания задач в данной области.

Язык проектирования программ включает определенный *внешний синтаксис*, описывающий логику программы при проектировании.

язык **PDL** содержит неопределенный *внутренний синтаксис*, который включает все структуры данных и процедуры по их обработке.

Язык **PDL** включает шесть групп операторов.

**Оператор выбора.**

а) **if** выражение;

**then** оператор1;

**else** оператор2;

б) **do case** (выражение);

**Оператор цикла.**

а) **do while** (выражение);

б) **do** переменная = выраж1 **to** выраж2 **by** выраж3;

**Оператор описания данных.**

**declare** имя атрибуты;

**Другие операторы.**

а) переменная = выражение;

б) **call** имя процедуры(список аргументов);

в) **return** (значение);

г) имя **procedure** (список параметров);

список операторов;

**end**;

д) **get** (список данных для ввода);

е) **put** (список данных для вывода);

**Оператор leave.**

Оператор **leave** обеспечивает выход из цикла, организованного с помощью оператора **do**. Оператор **leave** является типом управляющего оператора перехода.

**Предложения на естественном языке.**

Кроме указанных пяти классов операторов, любое предложение, написанное на естественном языке, можно использовать как оператор языка **PDL**:

а) Найти наибольший элемент в массиве B;

б) **do** для всех X из {a, b, c};

в) A = первый элемент B, который больше чем C.

1. **Нисходящее проектирование и нисходящая разработка.**

При нисходящем проектировании вначале проектируется управляющая программа — драйвер.

Затем более подробно представляются каждый из операторов псевдокода и разрабатываются другие модули.

При нисходящей разработке пользователь видит взаимодействие верхних уровней системы на начальных этапах. Изменения в этот период можно вносить относительно легко. Такими же свойствами обладает и *метод последовательной модификации* (*модернизации*).

1. **Пошаговое совершенствование.**

Нисходящее проектирование также называют *пошаговым совершенствованием*: программы иерархически структурируются и разбиваются путем последовательного уточнения. Накаждом шаге функционирование модуля описывается с помощью ссылок на предыдущие более подробные шаги.

*метод последовательной модификации* (*модернизации*). При использовании этого метода вначале проектируется и реализуется некоторый вариант системы.Пользователь очень быстро получает работающую систему.Процесс модернизации с последующим расширением функций

системы продолжается до тех пор, пока не будет получена окончательная версия.

1. **Восходящее проектирование.**

При *восходящем проектировании* вначале проектируются программы нижнего уровня. Обычно такой подход используется при проектировании операционных систем, где самым ниж-ним уровнем иерархии являются аппаратные средства (технология виртуальных машин).

1. **Подыгрывающие программы (заглушки).**

Нисходящее кодирование. В этом случае в первую очередь проверяют модули управляющей программы, а также модули A, B, C. Пользователь системы проверяет функционирование верхнего уровня на начальном этапе разработки, поэтому сделать любые необходимые изменения в спецификациях гораздо легче.

Единственное неудобство при таком методе кодирования заключается в том, что для проверки модулей A, B, C требуются также модули АА, АВ, АС, ВА, ВВ и СА. Для этих целей служат *подыгрывающие программы — заглушки*.

1. **Структурное проектирование. Простая программа. Элементарная программа. Управляющие структуры, способы их описания.**

Одним из эффективных методов разработки программ является метод пошагового совершенствования. Использование **PDL** хорошо согласуется с этим методом. Программист обду-

мывает проект задачи все более детально, причем каждый шаг является «интеллектуально управляемой» компонентой задачи.

Вначале программист представляет задачу как набор задач:

**do** task A;

**do** task B;

**do** task C;

Каждая из задач определяется и детализируется с помощью спецификаций. Каждую небольшую задачу можно представить в виде нескольких предложений **PDL**, входящих в некоторую процедуру. Если задача сложная, ее можно представить, как отдельную процедуру.

Для формализации процесса нисходящей разработки вводится понятие «простая программа».

*Простая программа* определяется как программа, которую можно представить в виде

структурной схемы со следующими свойствами:

1) существуют только одна входная и одна выходная дуги;

2) для каждого узла существует путь от входной дуги через этот узел к выходной дуге.

Определим *элементарную программу* как простую программу, которая не включает простых программ, состоящих более чем из одного узла.

Если небольшое количество узлов объединено в элементарные программы, то понять их функции также относительно несложно. Конструкция **if** — **then** — **else** состоит из трех узлов (один предикат и две функции); для нее можно определить функцию, включающую в себя функции, соответствующие частям **then** и **else** оператора. Изучение сложных программ основано на объединении сведений о более мелких составных частях программы.

Для структурного проектирования обычно используются следующие операторы: **if** — **then** — **else**, **while** — **do**, последовательность. Указанные управляющие структуры помогают программистам создать простые программы (функцию, которая преобразует входные данные в выходные).

В языке **PDL** используются элементарные подпрограммы с наименьшим числом узлов. Операторы **if** и **do while** являются минимальным набором, поскольку было доказано, что любая программная функция может быть представлена программой с указанными двумя управляющими структурами.

1. **Скалярные и агрегативные типы данных.**

Для облегчения процесса формализации задачи в языки программирования включены наборы различных типов данных.

Основным атрибутом переменной является ее *тип* или множество значений, которые может принимать переменная.

Переменные, объявленные как *элементарные* типы данного языка, называются *скалярными* переменными, а переменные, состоящие из наборов существующих типов данных, называются *агрегативными* переменными.

1. **Массивы. Структуры. Списки. Очереди. Стеки. Множества. Графы. Деревья.**

*Массивы* — это простейшие агрегативные данные в языках программирования. Массивом называется упорядоченный набор данных одного типа

Самой сложной разновидностью данных в языках программирования являются *структуры.* Структурой называют поименованную совокупность различных типов данных.

*Списком* называют упорядоченный набор переменных одного типа. (а как же Python?) Список отличается от массива тем, что его размер обычно является переменной величиной, т.е. элементы могут добавляться в список и изыматься из него.

*Очередь* — это упорядоченный список, в один конец которого элементы добавляются, а из другого изымаются. Очередь называют списком FIFO — Fist In, Fist Out (поступивший первым обслуживается первым).

*Стек* — это упорядоченный список, в один конец которого элементы добавляются и изымаются из этого же конца. Стек называют списком LIFO — Last In, Fist Out (поступивший последним обслуживается первым).

*Множества* — это совокупность переменных одного типа. Множество аналогично списку, за исключением того, что порядок расположения элементов не имеет значения.

*Направленный граф —* это структура, состоящая из *узлов* и *дуг,* причем каждая дуга направлена от одного узла к другому.

Для ненаправленных графов дугам соответствуют два направления — вперед и назад.

*Дерево* — это направленный граф, обладающий следующими свойствами:

1) только один узел не имеет дуг, входящих в него (корневой узел);

2) в каждый узел входит не более одной дуги;

3) в каждый узел можно попасть из корневого узла за несколько шагов.

1. **Абстрактные конструкции.**

Принцип *информационной локализованности.* Этот принцип заключается в том, что вся информация о структуре данных сосредотачивается в одном модуле. Доступ к данным осуществляется из этого модуля. Таким образом, внесение изменения в структуру данных не сопряжено с особыми затруднениями, потому что при этом меняется только один модуль.

*Данные абстрактного типа* создаются с использованием принципа информационной локализованности.

В практике программирования используются два способа программирования.

по *управлению* (т.е. выбор решения «что делать дальше?»). В этом случае структура программы понятна, а влияние логики программы на данные неясно.

*модульное* программирование (программист создает отдельные модули, которые выполняют определенное количество операций). В этом случае для обращения к модулю достаточно его имени и определения функций, с помощью которых можно обращаться к этому модулю.

В настоящее время (в зависимости от возможностей языка) используются три основных способа создания данных абстрактного типа.

- *Фиксированные типы данных абстрактного типа*

*- Размещение указателей*

*- Защита данных от несанкционированного доступа*

1. **Фиксированные данные абстрактного типа.**

Данный подход ориентирован на использование языков, «плохо» поддерживающих сложные структуры данных.

программист определяет данные абстрактного типа как структуру данных и оформляет каждую операцию с такими данными в виде отдельной процедуры; при этом со структурами

обращаются как с параметрами.

1. **Размещение указателей.**

Используя переменные типа указатель, можно построить конструкцию, близкую к фиксированным данным абстрактного типа.

Используя данные абстрактного типа, пользователь определяет данные как переменную — указатель и вызывает процедуру для размещения данных и обращения к этой структуре.

Рассмотренный способ удовлетворяет основным предъявляемым требованиям: имена структур и их представления разъединены, все операции со стеком сосредоточены внутри одной

процедуры (только из этой процедуры можно обращаться к данным). Однако данному способу присущ тот же недостаток, что и первому. С его помощью действительно строятся данные

абстрактного типа, однако во многих современных языках не существует ограничений, которые гарантировали бы, что к этим данным никто не обращается, т.е. невозможно запретить

программисту напрямую обратиться к данным, на которые ссылается указатель.

1. **Защита данных от несанкционированного доступа.**

Этот способ организации данных подобен предыдущему, но содержит средства контроля для определения некорректных обращений. Для определения данных этого типа добавляется

атрибут ENVIRONMENT (среда, окружение).

Заметим, что цель использования прав доступа заключается в ограничении доступа к абстрактным типам данных, а не в обеспечении безопасности системы.

**ПРАВИЛЬНОСТЬ ПРОГРАММ**

1. **Правильность программ.**

Одним из важных методов повышения эффективности проектирования программ является верификация программ или математическое доказательство того, что программа работает правильно.

Для доказательства правильности программ используется аксиоматический подход, при котором применяется теория перечисления предикатов. Предполагается, что каждый оператор в программе выполняет заранее определенные действия, зависящие только от синтаксиса языка.

Для двух предикатов *P* и *Q* и оператора *S* необходимо определить, истинно ли выражение:

«Если *P* истинно и если выполняется оператор *S*, то *Q* истинно».

*P**S**Q*, где *P* называется предусловием истинности *Q* после выполнения программы *S*.

Доказательство правильности программы заключается в определении, является ли выражение {*P*}*S*{*Q*} истинным относительно входных спецификаций *P*, выходных спецификаций *Q* и

операторов *S* программы. Если {*P*}*S*{*Q*} истинно, то это означает, что доказана правильность программы *S* относительно *P* и *Q*.

1. **Правила следствия.**

Для целей доказательства правильности программ к правилам исчисления предикатов следует добавить правила, необходимые для выполнения последовательности операторов программы. Эти правила, называемые *правилами следствия*, формулируются следующим образом:

1) если {*P*}*S*{*Q*} и *Q* *R*, то {*P*}*S*{*R*};

2) если {*Q*}*S*{*R*} и *P* *Q*, то {*P*}*S*{*R*}.

1. **Аксиома присвоения.**

Так как только оператор присвоения может изменять значение переменных, то для определения правильности оператора присвоения нужно добавить только одну аксиому, она называется *аксиомой присвоения* и формулируется следующим образом:

*аксиомой присвоения* и формулируется следующим образом:

В соответствии с этой аксиомой устанавливается, что если *P* — утверждение, содержащее переменную *x*, истинно, то *P* *должно быть истинно и до выполнения оператора присвоения,* *если x изменяется* expr.

1. **Аксиома следования.**

*Два оператора программы могут быть объединены.* Если после выполнения *S*1 предикат *Q* остается истинным и *Q* есть предусловие для оператора *S*2, то *P* является предусловием для операторов *S*1; *S*2.

1. **Аксиома цикла.**

Утверждается, что если истинность *P* не изменяется оператором *S* (т.е. является инвариантом), то *P* инвариантно относительно цикла, содержащего *S*. Кроме того, при выходе из цикла значение выражения *B* ложно.

**6. Аксиома выбора.**

Эти аксиомы устанавливают простые соотношения для оператора **if**.

**7. Правила целочисленной арифметики — коммутативность, ассоциативность, дистрибутивность, вычитание, обработка констант**.

**8. Доказательство правильности программ.**

* Доказательства длинны и сложны даже для простой программы.
* Если оперировать нецелочисленными данными, то формулировка аксиом становится затруднительной. Операции со строками, плавающей точкой, доступ к базам данных вызывают серьезные осложнения при аксиоматическом подходе.
* Даже для относительно простого случая — использования целочисленных данных — существуют трудности. Например, все ЭВМ используют слова фиксированной длины для записи целых чисел. Возможно переполнение разрядной сетки, и аксиомы должны учитывать эти условия.
* Реальные языки программирования имеют встроенные структуры, которые нелегко поддаются проверке на правильность.

Несмотря на отмеченные недостатки, доказательства правильности программ достаточно широко применяются. Способы проверки правильности программ можно использовать при проектировании реальных программ.

**ТЕСТИРОВАНИЕ**

1. **Определение процесса тестирования. Определение хорошего теста. Определение хорошего прогона.**

Программа тестируется не для того, чтобы показать, что она работает, а скорее наоборот — тестирование начинается с предположения, что в ней есть ошибки (это предположение справедливо практически для любой программы), а затем уже обнаруживается их максимально возможное число.

*Тестирование — это процесс исполнения программы с целью обнаружения ошибок.*

Термин «удачный» означает «результативный», а слово «неудачный» — «нежелательный», «нерезультативный».

Тестовый прогон, приведший к обнаружению ошибки, нельзя назвать неудачным хотя бы потому, что, как отмечалось выше, это целесообразное вложение капитала.

Тестовый прогон – «удачный», если в процессе его выполнения обнаружена ошибка, и «неудачный», если — корректный результат.

тестирование представляется деструктивным процессом попыток обнаружения ошибок в программе (наличие которых предполагается).

Набор тестов, способствующий обнаружению ошибки, считается удачным и наоборот.

1. **Тестирование программы как черного и белого ящика.**

**Стратегия черного ящика** - *тестирование с управлением по данным* или *тестированием с управлением по входу-выходу*. При использовании этой стратегии программа рассматривается как черный ящик.

Иными словами, такое тестирование имеет целью выяснение обстоятельств, в которых поведение программы не соответствует ее спецификации.

При таком подходе обнаружение всех ошибок в программе является критерием *исчерпывающего входного тестирования*. Последнее может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных.

**Стратегия *белого ящика****,* или стратегия тестирования, *управляемого логикой программы*, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы (к сожалению, здесь часто не используется спецификация программы).

**3. Принципы тестирования.**

*- Описание предполагаемых значений выходных данных или результатов должно быть необходимой частью тестового набора*.

Тест должен включать две компоненты:

- описание входных данных

- описание точного и корректного результата, соответствующего набору входных данных.

*- Следует избегать тестирования программы ее автором*.

- *Программирующая организация не должна сама тестировать разработанные ею программы*.

- *Необходимо досконально изучать результаты применения каждого теста*.

- *Тесты для неправильных и непредусмотренных входных данных следует разрабатывать так же тщательно, как для правильных и предусмотренных*.

- *Необходимо проверять не только, делает ли программа то, для чего она предназначена, но и не делает ли она то, что не должна делать*.

- *Не следует выбрасывать тесты, даже если программа уже не нужна*.

- *Нельзя планировать тестирование в предположении, что ошибки не будут обнаружены*.

- *Вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части*.

- *Тестирование — процесс творческий*.

*Тестирование — это процесс выполнения программ с целью обнаружения ошибок*.

*Хорошим считается тест, который имеет высокую вероятность обнаружения еще не выявленной ошибки*.

*Удачным считается тест, который обнаруживает еще не выявленную ошибку*.

1. **Технологии ручного тестирования. Инспекции исходного текста. Сквозные просмотры. Метод оценки программ посредством просмотра.**

Эксперименты показали, что методы ручного тестирования достаточно эффективны с точки зрения нахождения ошибок, так что один или несколько из них должны использоваться в каждом программном проекте.

Ручные методы способствуют существенному увеличению производительности и повышению надежности программы. Во-первых, они обычно позволяют раньше обнаружить ошибки, уменьшить стоимость исправления последних и увеличить вероятность того, что корректировка произведена правильно.

Во-вторых, психология программистов, по-видимому, изменяется, когда начинается тестирование на ЭВМ. Возрастает внутреннее напряжение, и появляется тенденция «исправлять ошибки так быстро, как только это возможно». В результате программисты допускают больше промахов при корректировке ошибок, уже найденных во время тестирования на ЭВМ, чем при корректировке

ошибок, найденных на более ранних этапах.

- Инспекции и сквозные просмотры:

Инспекции и сквозные просмотры включают в себя чтение или визуальную проверку программы группой лиц. Оба метода предполагают некоторую подготовительную работу. Завершающим этапом является «обмен мнениями» — собрание, проводимое участниками проверки. Цель такого собрания — нахождение ошибок, но не их устранение (т.е. тестирование, а не отладка).

- **Инспекции исходного текста:**

Инспекции исходного текста представляют собой набор процедур и приемов обнаружения ошибок при изучении (чтении) текста группой специалистов.

Инспектирующая группа включает обычно четыре человека, один из которых выполняет функции председателя.

Членами группы являются автор программы, проектировщик (если он не программист) и специалист по тестированию.

Инспекционное заседание разбивается на две части:

1. Программиста просят рассказать о логике работы программы.
2. Программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования

Время и место проведения инспекции должны быть спланированы так, чтобы избежать любых прерываний инспекционного заседания. Его оптимальная продолжительность, по-видимому, лежит в пределах от 90 до 120 мин.

Наконец, инспекция является способом раннего выявления наиболее склонных к ошибкам частей программы, позволяющим сконцентрировать внимание на этих частях в процессе выполнения тестирования на ЭВМ.

- **Сквозные просмотры:**

Сквозной просмотр, как и инспекции, представляет собой набор процедур и способов обнаружения, осуществляемых группой лиц, просматривающих текст программы.

Подобно инспекции, сквозной просмотр проводится как непрерывное заседание, продолжающееся один или два часа.

Группа по выполнению сквозного просмотра состоит из 3—5 человек.

Вместо того, чтобы просто читать текст программы или использовать список ошибок, участники заседания «исполняют роль вычислительной машины».

Лицо, назначенное тестирующим, предлагает собравшимся небольшое число написанных на бумаге тестов, представляющих собой наборы входных данных (и ожидаемых выходных данных) для программ или модуля. Во время заседания каждый тест мысленно выполняется. Это означает, что тестовые данные подвергаются обработке в соответствии с логикой программы.

Состояние программы (т.е. значения переменных) отслеживается на бумаге или доске.

тесты сами по себе не играют критической роли, скорее они служат средством для первоначального понимания программы и основой для вопросов программисту о логике проектирования и принятых допущениях.

* **Оценка посредством просмотра:**

Оценка посредством просмотра является методом оценки анонимной программы в терминах ее общего качества, ремонтопригодности, расширяемости, простоты эксплуатации и ясности.

Цель данного метода — обеспечить программиста средствами самооценки.

Выбирается программист, который должен выполнять обязанности администратора процесса. Администратор, в свою очередь, отбирает приблизительно 6—20 участников

(6 — минимальное число для сохранения анонимности).

Каждого участника просят представить для рассмотрения две свои программы — наилучшую (с его точки зрения) и низкого качества.

Отобранные программы случайным образом «распределяются между участниками. Им дается на рассмотрение по четыре программы. Две из них являются «наилучшими», а две — «наихудшими», но рецензенту не сообщают о том, какая программа к какой группе относится.

Каждый участник тратит на просмотр одной программы 30 мин и заполняет анкету для ее оценки. После просмотра всех четырех программ оценивается их относительное качество. В анкете для оценки проверяющему предлагается оценить программу по семибалльной шкале

(1 означает определенное «да», 7 — определенное «нет»)

Участники получают статистическую сводку, которая содержит общую и детальную классификацию их собственных программ в сравнении с полным набором программ, и

анализ того, насколько оценки чужих программ совпадают с оценками тех же самых программ, данными другими проверяющими.

Цель такого просмотра — дать возможность про граммистам самим оценить свою квалификацию.

1. **Принципы проектирования теста.**

*Какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения большинства ошибок?* Изучение методологий проектирования тестов дает ответ

на этот вопрос.

По-видимому, наихудшей из всех методологий является тестирование со случайными входными значениями (стохастическое)

Можно разработать довольно полный тест, используя определенную методологию проектирования, основанную на принципе черного ящика, а затем дополнить его проверкой логики программы (т.е. с привлечением методов белого ящика).

Методологии, обсуждаемые в настоящем разделе, представлены ниже:

Черный ящик Белый ящик

Эквивалентное разбиение Покрытие операторов

Анализ граничных значений Покрытие решений

Применение функциональных диаграмм Покрытие условий

Покрытие решений/условий

Предположение об ошибке Комбинаторное покрытие условий

при проектировании эффективного теста программы рекомендуется использовать если не все эти, то, по крайней мере, большинство из них, так как каждый из них имеет определенные достоинства и недостатки (например, возможность обнаруживать и пропускать различные типы ошибок).

1. **Технологии тестирования по принципу белого ящика. Покрытие операторов. Покрытие решений. Покрытие условий. Покрытие решений/условий. Комбинаторное покрытие условий.**

**- тестирование по принципу белого ящика –** характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы.

Исчерпывающее тестирование по принципу белого ящика предполагает выполнение каждого пути в программе; но поскольку в программе с циклами выполнение каждого пути обычно нереализуемо, то тестирование всех путей не рассматривается как перспективное.

- ***Покрытие операторов*** - критерием покрытия является выполнение каждого оператора программы, по крайней мере, один раз. К сожалению, это слабый критерий, так как выполнение каждого оператора, по крайней мере, один раз есть необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика.

- ***Покрытие решений*** - Более сильный критерий покрытия логики программы известен как *покрытие решений,* или *покрытие переходов.*

Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение *истина* и *ложь*, по крайней мере, один раз. Иными словами, каждое направление перехода должно быть реализовано, по крайней мере, один раз.

Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом значение *истина* или *ложь* и что каждой точке входа должно быть передано управление при вызове программы, по крайней мере, один раз.

- ***Покрытие условий*** -Лучшим критерием по сравнению с предыдущим является

*покрытие условий.*

В этом случае записывают число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз.

Поскольку, как и при покрытии решений, это покрытие не всегда приводит к выполнению каждого оператора, к критерию требуется дополнение, которое заключается в том, что каждой

точке входа в программу или подпрограмму, а также ON-единицам должно быть передано управление при вызове, по крайней мере, один раз.

**Покрытие решений/условий** - Он требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере,

один раз, все результаты каждого решения выполнялись, по крайней мере, один раз и каждой точке входа передавалось управление, по крайней мере, один раз.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий;

***Комбинаторное покрытие условий*** *-* Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и во всех точках входа выполнялись, по крайней мере, один раз.

1. **Технологии тестирования по принципу черного ящика.**

Тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных.

Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

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

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

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как *эквивалентное разбиение.*

Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое — для разработки минимального набора тестов, покрывающих эти условия.

1. **Эквивалентное разбиение. Способы формирования классов эквивалентности. Правила создания тестов по классам эквивалентности.**

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

1) выделение классов эквивалентности

2) построение тестов.

***Выделение классов эквивалентности*** - Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. Для проведения этой операции используют таблицу

|Входные условия | Правильные классы эквивалентности | неправильные классы эквивалентности |

**Правильные классы** эквивалентности, представляют правильные входные данные программы

**Неправильные классы** эквивалентности, представляющие все другие возможные состояния условий (т.е. ошибочные входные значения).

Правила:

1. Если входное условие описывает *область* значений (например, «целое данное может принимать значения от 1 до 999»), то определяются:

−один правильный класс эквивалентности (значений от 1 до 999);

−два неправильных класса (значения целого данного <1 и значение целого данного >999).

2. Если входное условие описывает *число* значений (например, «в автомобиле могут ехать от одного до шести человек»), то определяются один правильный класс эквивалентности и два неправильных (ни одного и более шести).

3. Если входное условие описывает множество входных значений и есть основание полагать, что каждое значение программа трактует особо (например, «известны способы передви-

жения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, ПЕШКОМ или МОТОЦИКЛЕ»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс

эквивалентности (например, «НА ПРИЦЕПЕ»).

4. Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ — буква) и один неправильный (первый символ — не буква).

5. Если есть любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс эквивалентности разбивается на меньшие классы эквивалентности.

***Построение тестов*** -Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.

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

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

Хотя эквивалентное разбиение значительно лучше случайного выбора тестов, оно все же имеет недостатки (т.е. пропускает определенные типы высокоэффективных тестов).

1. **Анализ граничных значений.**

Как показывает опыт, тесты, исследующие *граничные условия*, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия — это ситуации, возникающиенепосредственно на границах, а также выше или ниже границвходных и выходных классов эквивалентности.

Анализ граничных значений исследует ситуации, возникающие *на границах и вблизи* *границ эквивалентных разбиений.*

Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

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

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

приведем несколько правил этого метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области
2. Построить тесты для минимального и максимального значения условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений.
3. Использовать правило 1 для каждого выходного условия. Заметим, что важно проверить границы пространства результатов
4. Использовать правило 2 для каждого входного условия.
5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.
6. Попробовать свои силы в поиске других граничных условий.

Анализ граничных значений, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако он часто оказывается неэффективным из-за того, что внешне выглядит простым. Необходимо понимать, что граничные условия могут быть едва уловимы и, следовательно, определение их связано с большими трудностями.

1. **Применение функциональных диаграмм. Способы задания ограничений на вход и выход. Технология построения функциональных диаграмм. Технология построения таблицы решений. Формирование тестов по таблице решений.**

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют *комбинаций* входных условий.

Метод функциональных диаграмм или диаграмм причинно-следственных связей помогает систематически выбирать тесты с высокой результативностью. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.

Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке.

Построение тестов этим методом осуществляется в несколько этапов.

1. Спецификация разбивается на «рабочие» участки.
2. В спецификации определяются причины и следствия. *Причина* есть отдельное входное условие или класс эквивалентности входных условий. *Следствие* есть выходное условие или преобразование системы (остаточное действие, которое входное условие оказывает на состояние программы или системы). Причины и следствие определяются путем последовательного (слово за словом) чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждой причине и следствию приписывается отдельный номер.
3. Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф, связывающий причины и следствия. Это и есть функциональная диаграмма.
4. Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными изза синтаксических или внешних ограничений.
5. Путем методического прослеживания состояний условий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.
6. Столбцы таблицы решений преобразуются в тесты.

Вероятная ошибка в документации: стр. 149.

Для иллюстрации изложенного рассмотрим диаграмму, отображающую спецификацию:

Символ(ы) в колонке(ках) 1(1,2) должен(ы) быть буквой(вами) «А» или «В», а в

колонке 2(3) — цифрой. В этом случае файл обновляется. Если

первый символ неправильный, то выдается сообщение Х12, а

если второй символ неправильный — сообщение X13.

Причинами являются: 1 — символ «А» в колонке 1; 2 —

символ «В» в колонке 1(2); 3 — цифра в колонке 2(3), а следствиями —

70 — файл обновляется; 71 — выдается сообщение Х12; 72 —

выдается сообщение X13.

Вероятная ошибка в документации: стр. 159 – 160.:

Рис. 6.20 приведен в качестве примера функциональной

диаграммы…..

ошибочно вычислены состояния узлов 5,6:

Таким образом, возможны пять(три) состояний узлов 1—4, например значения

0 0 0 0 (5 = 0, 6 = 0),

1 0 0 0 (5 = 1, 6 = 0),

1 0 0 1 (5 = 1, 6 = 0), => 1 0 0 1 (5 = 1, 6 = 1),

1 0 1 0 (5 = 1, 6 = 0), => 1 0 1 0 (5 = 1, 6 = 1),

0 0 1 1 (5 = 0, 6 = 1),

**Замечания**.

Применение функциональных диаграмм — систематический метод генерации тестов, представляющих комбинации условий. Альтернативой является специальный выбор комбинаций, но при этом существует вероятность пропуска многих «интересных» тестов, определенных с помощью функциональной диаграммы.

При использовании функциональных диаграмм требуется трансляция спецификации в булевскую логическую сеть. Следовательно, этот метод открывает перспективы ее применения и дополнительные возможности спецификаций. Действительно, разработка функциональных диаграмм есть хороший способ обнаружения неполноты и неоднозначности в исходных спецификациях.

**11. Предположение об ошибке.**

Человек, обладающий практическим опытом, часто подсознательно применяет метод проектирования тестов, называемый предположением об ошибке. При наличии определенной программы он интуитивно предполагает вероятные типы ошибок и затем разрабатывает тесты для их обнаружения.

Процедуру для метода предположения об ошибке описать трудно, так как он в значительной степени является интуитивным. Основная идея его заключается в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка написать тесты.

Поскольку данная процедура не может быть четко определена, лучшим способом обсуждения смысла предположения об ошибке представляется разбор примеров.

1. **Стратегия тестирования.**

Методологии проектирования тестов, могут быть объединены в общую стратегию.

1. Если спецификация содержит комбинации входных условий, то начать рекомендуется с применения метода функциональных диаграмм.
2. В любом случае необходимо использовать анализ граничных значений.
3. Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущих шагах.
4. Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке.
5. Проверить логику программы на полученном наборе тестов. Для этого нужно воспользоваться критерием покрытия решений, покрытия условий, покрытия решений/условий либо комбинаторного покрытия условий (последний критерий является более полным).

**ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММ**

1. **Стандартные методы проектирования.**

- Разбиение задачи на независимые подзадачи.

- Разбиение задачи на одинаковые по сложности части.

**-** Рекурсия.

- Динамическое программирование.

- Моделирование.

- Поиск. Поиск в списках. Прямой поиск. Линейный поиск.

- Поиск с возвратом.

- Стратегия распределения памяти.

- Алгоритм выбора из конечного состояния.

- Сопрограммы.

1. **Разбиение задачи на независимые подзадачи.**

Основным алгоритмом, используемым для решения задач, является алгоритм разбиения на независимые подзадачи. Для того, чтобы решить задачу A, ее первоначально необходимо разбить на независимые подзадачи B, C, D и т.д.

1. **Разбиение задачи на одинаковые по сложности части.**

Данный подход означает разделение задачи на подзадачи, равные примерно по сложности, для того чтобы перейти к решению значительно более простых задач, чем первоначальная.

Каждая из частей может иметь рекурсивное решение.

1. **Рекурсия.**

Иногда возможно сформулировать решение задачи как известное преобразование другого, более простого, варианта той же задачи. Если этот процесс продолжить до тех пор, пока не получится нужное значение, то подобное решение называется рекурсивным.

1. **Динамическое программирование.**

Динамическое программирование — это табличный метод, при котором используется одна и та же схема вычисления всех подзадач данной задачи. При использовании этого метода однажды найденный результат записывается в таблицу и далее повторно не вычисляется.

1. **Моделирование.**

Строится представление определенных свойств задачи, называемой моделью, а все другие свойства не учитываются. При этом проводится анализ поведения нужных свойств модели.

Такой подход называется моделированием.

Данный подход заслуживает пристального внимания программистов при разработке больших программных комплексов.

**7. Поиск. Поиск в списках. Прямой поиск. Линейный поиск.**

Поиск — это процесс нахождения нужных значений в некотором множестве.

Обычно каждый элемент имеет три компонента:

−ключ — это элемент, используемый для доступа к данным (он может быть именем в справочнике, словом виндексе или названием раздела в таблице);

−аргумент — это числа, связанные с элементом (адрес, номер телефона и др.);

−структуру — это дополнительная информация, принимаемая для описания данных, такая, как адрес следующего элемента в семействе и т.п.

- Поиск в списках:

Данный поиск проводится четырьмя различными способами: прямым, линейным, двоичным и хешированием.

* Прямой поиск. При прямом поиске местоположение элемента определяется непосредственно с помощью ключа (комната в здании). Прямой поиск применяется тогда, когда число элементов фиксировано и их сравнительно немного.
* Линейный поиск. Данный вид поиска наиболее простой для программной реализации, хотя его выполнение занимает много времени. Элементы проверяются последовательно, по одному, пока нужный элемент не будет найден.
* Двоичный поиск. Для ведения двоичного поиска ключи должны быть упорядочены по величине или алфавиту.
* Хеш-поиск. Этот вид поиска является попыткой применить прямой поиск для поиска в больших наборах данных. Из изначального значения ключа вычисляется значение псевдо ключа, называемого хеш-кодом, и этот код используется как индекс в таблице адресов элементов. Если все ключи соответствуют разным хеш-кодам, то любой элемент будет найден за один прямой поиск.

1. **Поиск с возвратом.**

Если данные организованы как структура дерева, алгоритм поиска сводится к алгоритму, просматривающему все узлы дерева. Для ведения поиска существуют два основных алгоритма поиска: поиск в глубину и поиск в ширину.

* Поиск в глубину. При поиске в глубину каждая ветвь дерева просматривается слева направо.
* Поиск в ширину. Это алгоритм поиска, при котором просматривается каждый уровень в направлении сверху вниз

1. **Стратегия распределения памяти.**

Одним из типов поиска являются задачи по распределению памяти (наиболее характерны они при создании ОС), однако во многих прикладных алгоритмах эти задачи возникают при создании прикладных программ, работающих с большим объемом данных. Если до начала обработки полный объем данных, которые подлежат обработке, неизвестен, то используется общий подход для распределения областей этой памяти согласно заданным требованиям.

* Первое возможное размещение. Последовательно просматриваются области памяти, пока не найдется первая, достаточная для размещения.
* Наилучшее размещение. Последовательно просматриваются все области, выбирается наименьшая область, размер которой больше или равен требуемому объему для размещения данных.
* Сопрягаемые области памяти*.* Областями памяти являются *N* цепочек размером 2*N* каждая. Если область размером 2*K* отсутствует, а имеется свободная область размером 2*K*+1, то она разбивается на две сопрягаемые области размером 2*K*. После того как области освободятся, они рассматриваются как области размером 2*K*, которые можно объединить в область размером 2*K*+1.

1. **Алгоритм выбора из конечного состояния.**

Часто задача может быть сведена к множеству действий, зависящих от текущего состояния программы. Такой способ решения задачи называется выбором из конечного состояния и

обычно включает таблицу, описывающую выполняемые действия. Строки таблицы определяют состояние программы, а столбцы — возможные действия. Элементы таблицы описывают выполнение возможных действий, в частности имя вызываемой подпрограммы для обработки действий.

Решение подобных задач строится по типу модели конечного автомата.

1. **Сопрограммы.**

Сопрограмма — это такой вид программы, который сохраняет текущее состояние счетчика команд. Когда программа вызывается повторно, выполнение продолжается с адреса, записанного в счетчике программ, а не с начала программы.

**МЕТОДЫ УПРАВЛЕНИЯ ПРОЕКТИРОВАНИЕМ ПРОГРАММНЫХ ИЗДЕЛИЙ**

Основная цель управления — организовать и связать взаимодействие исполнителей при создании программного продукта.

1. **Понятие изделия как средства общения.**

Действия при создании программного изделия по многим параметрам совпадают с действиями при создании технического изделия. То есть самые общие действия при создании изделия име-

ют аналоги при создании программного продукта, таким образом, можно использовать опыт по управлению изготовлением.

1. **Нисходящий анализ процесса управления созданием программного изделия.**

Управление проектированием программного изделия включает в себя следующие функции:

−планирование;

−разработку;

−обслуживание;

−выпуск документации;

−испытания;

−поддержку;

−сопровождение;

Каждая организация должна иметь администратора (директора), именно он несет ответственность за успех и неудачу разработки. В структуре также должно существовать то или иное важное направление деятельности, включая функцию разработки. Лицо, которое руководит этим направлением, считается ответственным за все аспекты создания изделия, выпускаемого организацией. Чтобы координировать процесс разработки, это лицо имеет право назначать администраторов изделия и руководителей проектов и обеспечивать их взаимодействие.

Важнейшим принципом любого вида управления является разделение целого на части, и многие методы и средства основываются именно на этом принципе.

Совокупность точек, в которых две функциональные группы взаимодействуют друг с другом, называется *организационной границей*.

Основное свойство организационной границы состоит в разграничении ответственности (кто и что делает, каким образом, для чего и т.д.). Неполное определение, двусмысленность и сложность приводят к невозможности описания, а следовательно, и понимания природы взаимодействия.

Пропорциональное распределение ответственности обеспечивается соответствующими стратегиями управления. В них обязательно должны предусматриваться возможности невыполнения взятых обязательств и учитываться поправки на исключительные случаи, чтобы сделать планы и процедуры жизнеспособными.

1. **Установление целей и средства их достижения.**

Первым шагом процесса установления и достижения целей является подбор необходимого персонала.

Основным методическим принципом управления разработкой является *целевое управление.*

Целевое управление представляет собой концепцию планирования и управления, с помощью которой руководитель устанавливает соответствующие цели через своего непосредственного руководителя более высокого ранга и участвует в установлении целей последнего; при этом результаты его деятельности оцениваются на основании конкретного обсуждения и документального рецензирования.

Основной план для программного изделия — **соглашение о требованиях**.

Цели, сформулированные в этом плане, должны включаться в индивидуальные рабочие планы, служащие тем механизмом, посредством которого в системе целевого управления достигается договоренность между исполнителями и их руководителями.

Соглашение о требованиях, план поддержки, распределение бюджета и другие средства устанавливают те границы, в пределах которых не требуется использование обратной связи.

Существует три вида основных критериев оценки эффективности той или иной деятельности:

−конкретные свойства;

−затрачиваемое время;

−стоимость.

1. **Подбор и обучение кадров.**

Поскольку разработка программных средств является достаточно сложной процедурой, то для ее реализации нужны специалисты высокой квалификации, т.е. для проектирования программного обеспечения необходимо выделять высококвалифицированный персонал на все участки работы. Следует искать таких людей, которые выполняли подобные функции достаточно хорошо, либо тех, кто выполнял очень грамотно функции чуть меньшей сложности.

Основная черта, которой должен обладать кандидат, — способность подчиняться дисциплине. Он

должен понимать важность *нисходящего подхода* в создании программного изделия и документирования программы до ее кодировки.

Важным фактором для успешного выполнения работ является обеспечение продвижения по службе.

Кроме вертикального продвижения по служебной лестнице, следует обеспечить продвижение по горизонтальному уровню. Должна поощряться широта профессиональных интересов и упрощаться проблема подбора кадров для выполнения менее важных функций путем включения в должностные инструкции более высоких уровней описания тех функций, выполнения которых достаточно для занятия более низких должностей в тех или иных подразделениях.

Принимая на работу людей любого уровня квалификации, необходимо обеспечить им возможность продвижения по службе в результате дополнительного профессионального обучения.

Не менее важно обучение вне рамок организации. Необходимо максимально использовать участие сотрудников проекта в семинарах по профилю, проводимых передовыми школами программирования. Естественно, что такое обучение дело дорогостоящее, поэтому необходимо иметь твердую уверенность, что затраты на обучение дадут должный эффект.

**5. Организация планирования разработки программного изделия. Виды планов.**

**Декомпозиция планов.**

Планы создания программного изделия должны охватывать этапы разработки, документирования, испытаний, обучения пользователей, сопровождения. Отсутствие планов — основная причина переделки программ.

Программное изделие — это собственно программа плюс документация, гарантия качества, рекламные материалы, обучение, распространение и сопровождение. Таким образом, для создания программного изделия требуется совокупность планов, охватывающих все аспекты разработки изделия и их связь с внешней средой.

Самым главным охватывающим планом является **целевая программа**. Цель ее максимизировать доход от капиталовложений. Целевая программа отвечает на вопрос, для чего предназначена та или иная деятельность. В целевой программе не сообщается, как цель должна быть достигнута, и редко устанавливаются сроки реализации. Обычно целевые программы редко имеют самостоятельную значимость, они предназначены для других программ, чтобы ответить на вопрос *зачем?*

Следующий уровень планирования — **стратегический план**. Он отвечает на вопросы *что?* И *когда?* При этом чем более конкретна цель, тем более своеобразны будут стратегии ее достижения.

Более конкретными являются **тактические планы**. Тактические планы отвечают на вопросы *кто?* и *где?*, т.е. конкретизируют способы достижения цели.

**Стратегический** план формулирует, что необходимо сделать для достижения цели и когда. **Тактический** план показывает, кто будет выполнять работу, как и когда.

Обычно целевая программа, стратегический и тактический планы редко существуют как самостоятельные элементы. Однако в тех случаях, когда формируется совокупность планов,

необходимо классифицировать каждый из них в соответствии с тремя указанными категориями, чтобы облегчить понимание задач.

Индивидуальные планы работ — это инструмент целевого управления, обеспечивающий фиксацию конкретных выработанных целей и сроков окончания работ, контроль за их завершением или прекращением работ. То есть он аккумулирует в себе целевой, стратегический и тактический планы на основе бюджета. Это действенное средство «одушевления» разработки программного изделия и выявления в ней роли конкретных участников.

Сетевой график является преимущественно тактическим планом. Он определяет в терминах заданий (работ) — как можно достигнуть цели и кто отвечает за выполнение конкретных

работ. Это также и стратегический план, в котором устанавливаются даты окончания работ.

При планировании (создании планов) используется схема нисходящего планирования. Эта процедура называется декомпозицией планов:

−план создания семейства программных изделий;

−план создания серии;

−план создания совокупности изделий;

−план производства конкретного изделия.

Законченность плана означает, что рассмотрены все необходимые вопросы и в то же время обеспечен необходимый уровень детализации.

Все имеющие отношение к делу вопросы должны обсуждаться на самом высоком уровне детализации и на самом низком уровне иерархии.

В процессе декомпозиции планов в качестве основного принципа должен служить следующий принцип: «Какая свобода выбора желательна на следующем шаге планирования?».

Обычно последующие этапы планирования выполняются на все более низких уровнях, и достаточно учитывать лишь область их полномочий.

Структура всех реальных иерархических планов обязательно должна совпадать.

Второй принцип декомпозиции — определить ограничения для следующего, более низкого, уровня «Сделай транслятор с языка за один год, потратив не более 300.000$». Здесь имеется

набор трех ограничений:

транслятор должен обрабатывать инструкции конкретного языка;

для создания отведено не более года;

бюджет — не более 300.000$.

Никаких других ресурсных ограничений нет. В этих рамках есть свобода выбора. Если продолжить декомпозицию плана, то возникнут все большие и большие ограничения, так что

на самом низком уровне будет обеспечена их абсолютная детализация. Но при этом, если полномочия распределены правильно, ни одно из первоначальных ограничений не будет нарушено, т.е. должна существовать обратная связь, в случае если потребуется изменение плана за пределами своих полномочий.

1. **Организационная структура группы планирования.**

В любой разрабатывающей организации должно быть создано специальное подразделение,

единственной обязанностью которого является планирование и управление организацией в целом.

Оно необходимо потому, что кто-то должен гарантировать существование планов для всех

подразделений, обеспечивать их совместимость и взаимную увязку. Люди, выполняющие эти функции, не должны одновременно иметь никаких других обязанностей. Планы подобны законам. Они должны выполняться неукоснительно.

Должностные лица, руководящие выполнением планов, называются администраторами планирования.

Группа планирования должна быть организована по принципу предметно-производственной специализации.

В должностных инструкциях должна быть четко определена одна-единственная точка соприкосновения, где каждая функциональная группа взаимодействует с группой планирования.

В большинстве организаций роль администратора планирования выполняет руководитель целевой программы.

Планирование должно опираться на гарантии правильного применения принципов конфигурационного управления. Для этой цели вводятся функции контроля документации, включая процедуры хранения и распространения проектной документации.

Вместе с тем планирование невозможно только по иерархии сверху. Поэтому каждая функциональная группа должна иметь собственный персонал тактического планирования. Их

взаимодействие должно строиться по схеме:

−подразделение планирования отвечает за целевое и стратегическое планирование, управление бюджетом и планами для всей организации;

−функциональная группа отвечает за тактическое планирование.

1. **Виды планов, связанных с созданием программного изделия.**

При использовании нисходящего планирования осуществляется постепенный переход от общего к частному (от целевой программы к стратегическим и тактическим планам) и от глобального к локальному (от плана семейства изделий, через план выпуска серии и совокупности изделий, к проекту конкретного программного изделия). Для облегчения понимания структуры различных планов вводятся «стандартные» определения.

Программное изделие — совокупность отдельных программных средств, их документации, гарантии качества (ISO 9000), рекламных материалов, мер по обучению пользователей, распространению и сопровождению программного обеспечения.

Совокупность изделий — это группа изделий, имеющих одну или несколько общих характеристик и работающих совместно в некоторой комбинации (ОС, компиляторы, сервисные программы, средства диагностики, которые управляются операционной системой, составляют такую группу).

Серия изделий — это сочетание аппаратных и программных средств, которые имеют одну и более общих связей и функционируют совместно в некоторой комбинации как самостоятельная система.

Семейство изделий — несколько связанных программных изделий, которые необязательно должны иметь какой-либо общий интерфейс и работать на одной и той же аппаратуре.

Планом самого высокого уровня является план выпуска семейства или серии изделий. Так как в их названии присутствует слово изделие — это стратегический план.

Он содержит элементы стратегии:

−как обеспечить совместимость с конкурирующими изделиями, благоприятствующую проникновению на рынок;

−периодичность усовершенствования в целях продления цикла жизни изделия и т.д.

Как только план серии одобрен, разрабатываются планы выпуска совокупности изделий.

**Распределение бюджета** - этот документ реализует концепцию приростного финансирования, он обеспечивает контроль за выполнением планов.

Первая задача решается после распределения бюджета, формируется план, который называется **соглашением о требованиях**

Как только разработка становится реальностью, формируются планы отдельных групп: группы испытаний, план выпуска документации, план обеспечения поддержки..

Там, где существует хотя бы небольшая вероятность неудачи, план должен предусматривать возможность создания экспериментального изделия.

Опытный образец — это почти законченное изделие. Он включает в себя все компоненты завершенного изделия, в том числе и черновой вариант технической документации, и предполагает проведение небольшого объема независимых испытаний с целью обеспечения необходимых условий для выработки критических замечаний по доработке изделия и гарантий минимального риска пользователя.

1. **Организация планирования разработки программного изделия.**

* **Организация планирования в фазе исследования**

Фаза исследований начинается тогда, когда подтверждается необходимость создания программного изделия, и заканчивается тогда, когда утверждены технические требования

* **Организация планирования в стадии анализа осуществимости**

В момент времени, когда ресурсы распределены (начало фазы осуществимости), но соглашения о требованиях еще нет (конец фазы осуществимости), новое изделие рассматривается с учетом будущих условий его использования. Для этого существует два средства: **конфигуратор** и **план выпуска**.

В **конфигураторе** перечисляются программные изделия, которые должны функционировать совместно с проектируемым изделием, т.е. Конфигуратор — это план создания интерфейсов высокого уровня, определяющих связи и способствующих взаимодействию функциональных групп, совместно разрабатывающих некоторое изделие.

**План выпуска** изделия имеет то же значение, что и конфигуратор, но охватывает только те элементы, которые должны быть готовы к определенному времени.

Последней задачей, выполняемой группой планирования в фазе осуществимости, является рассмотрение и утверждение соглашения о требованиях к программному изделию.

Это самый важный момент в планировании, т.к. он устанавливает направление разработки и инициирует начало разработки.

* **Организация планирования в фазах конструирования и кодирования**

В течение всей **фазы конструирования** (проектирования) группа выпуска документации и группа испытаний готовят планы издания документов и планы испытаний.

В течение фазы кодирования группа поддержки готовит свой план, а группа планирования рассматривает его в том же порядке, в котором до этого рассматривались план изданий и план испытаний.

* **Организация планирования в фазах оценки и использования**

Следующий период жизненного цикла программного изделия связан с принятием решения о целесообразности широкого распространения изделия. Рекомендации дает группа испытаний, а если решение о выпуске принято без ее согласия, то группа испытаний дает заключение о степени готовности изделия.

Для принятия решения о распространении изделия всегда требуется согласие группы планирования, потому что после выпуска изделия очень трудно исправить ошибки не только в программах, но и в документации.

Обычно группа поддержки настаивает на возможно более раннем выпуске изделия, чтобы своевременно закончить операции по сбыту или сохранить того или иного заказчика. Группа разработки выдвигает то же самое требование из практических соображений. Группа же испытаний обычно хочет продлить испытания. В этих условиях компромиссное решение принимает лишь группа планирования, как группа, отвечающая за комплексную увязку всех вопросов.

Группа планирования осуществляет текущий контроль за изделием в фазе использования, непрерывно наблюдая за уведомлениями о дефектах и запросами на расширение.

Именно администратор группы планирования отвечает за организацию устранения замеченных дефектов и целесообразность создания новых версий с расширенными возможностями. При этом его основная задача — постоянно снижать уровень поддержки и

сопровождения.

Однако независимо от первоисточника рекомендации, последнее слово принадлежит группе планирования. Никакая другая группа не имеет такого круга обязанностей, который необходим для ответственного принятия подобного решения.

**9. Вопросы, рассматриваемые в фазовых обзорах группой планирования.**

**10. Управление проектом.**

**11. Организация работы группы разработки в фазах создания программного изделия.**

**12. Организация работы группы обслуживания в фазах создания программного изделия.**

**13. Организация работы группы выпуска документации в фазах создания программного изделия.**

**14. Организация испытаний программного изделия.**