**54. Отладка ПО.**

Хотя слова «отладка» и «тестирование» часто используются как синонимы, под ними подразумеваются разные виды деятельности. Тестирование — это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ; отладка направлена на установление точной природы известной ошибки, а затем — на исправление этой ошибки. Эти два вида деятельности связаны — результаты тестирования являются исходными данными для отладки.

Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПО ошибки, поиска места ошибки в программах и документации ПО и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование. Поэтому важно составлять хорошие тесты для успешного выявления ошибок и их исправления.

Место отладки в цикле разработки программы:

Типичный цикл разработки, за время жизни программы многократно повторяющийся, выглядит примерно так:

1. Программирование — внесение в программу новой функциональности, исправление ошибок в имеющейся.
2. Тестирование (ручное или автоматизированное; программистом, тестером или пользователем; «дымовое», в режимечерного ящика илимодульное...) — обнаружение факта ошибки.
3. Воспроизведение ошибки — выяснение условий, при которых ошибка случается. Это может оказаться непростой задачей при программировании параллельных процессов и при некоторых необычных ошибках.
4. **Отладка** — обнаружение причины ошибки.

Выявленные программные ошибки, как правило, делятся на три вида:

1. ***Синтаксическая ошибка*** *.* Неправильное употребление синтаксических конструкций.
2. ***Семантическая ошибка*** *.* Нарушение семантики той или иной конструкции, например передача функции параметров, не соответствующих ее аргументам.
3. ***Логическая ошибка*** *.* Нарушение логики программы, приводящее к неверному результату. Это наиболее трудный для "отлова" тип ошибки, ибо подобного рода ошибки, как правило, кроются в алгоритмах и требуют тщательного анализа и всестороннего тестирования.
4. Поскольку безошибочное программирование почти невозможно, а ручная отладка немыслима, необходимы средства поиска ошибки (иногда это не так просто) и ее исправления. В каждой современной системе программирования существует специальное средство отладки программ — *отладчик* (debugger), который позволяет в режиме интерпретации установить контрольные точки, выполнить отдельные участки программы и посмотреть результаты работы операторов.

Ниже приводятся рекомендации по организации отладки в форме заповедей.

*Заповедь 1*. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

*Заповедь 2*. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

*Заповедь 3*. Готовьте тесты как для правильных, так и для неправильных данных.

*Заповедь 4*. Документируйте пропуск тестов через компьютер; детально изучайте результаты каждого теста; избегайте тестов, пропуск которых нельзя повторить.

*Заповедь 5.* Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.

*Заповедь 6.* Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

**53. Тестирование ПО. Методы «Черного ящика»**

**Тестирование чёрного ящика** или **поведенческое тестирование** — стратегия (метод) тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта.

Под «чёрным ящиком» понимается объект исследования, внутреннее устройство которого неизвестно. Манипулируя только лишь со входами и выходами, можно проводить определенные исследования.

Принципы тестирования чёрного ящика

В этом методе программа рассматривается какчерный ящик (т.е. то, реализацию, внутреннее строение чего мы не знаем). Целью тестирования ставится выяснение обстоятельств, в которых поведение программы не соответствует спецификации. Для обнаружения всехошибок в программе необходимо выполнить исчерпывающее тестирование, то есть тестирование на всевозможных наборах данных. Для большинства программ такое невозможно, поэтому применяют разумное тестирование, при котором тестирование программы ограничивается небольшим подмножеством всевозможных наборов данных. При этом необходимо выбирать наиболее подходящие подмножества, подмножества с наивысшей вероятностью обнаружения ошибок.

### Свойства правильно выбранного теста

1. Уменьшает более, чем на одно число других тестов, которые должны быть разработаны для разумного тестирования.
2. Покрывает значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибки до и после ограниченного множества тестов.

### Приёмы тестирования чёрного ящика

1. Эквивалентное разбиение.
2. Анализ граничных значений.
3. Анализ причинно-следственных связей.
4. Предположение об ошибке.

Рассмотрим подробнее каждый из этих методов:

#### **Эквивалентное разбиение**

Основу метода составляют два положения:

1. Исходные данные необходимо разбить на конечное число классов эквивалентности. В одном классе эквивалентности содержатся такие тесты, что, если один тест из класса эквивалентности обнаруживает некоторую ошибку, то и любой другой тест из этого класса эквивалентности должен обнаруживать эту же ошибку.
2. Каждый тест должен включать, по возможности, максимальное количество классов эквивалентности, чтобы минимизировать общее число тестов.

Разработка тестов этим методом осуществляется в **два этапа**: выделение классов эквивалентности и построение теста. Классы эквивалентности выделяются путём выбора каждого входного условия, которые берутся с помощью технического задания или спецификации и разбиваются на две и более группы.

Выделение классов эквивалентности является эвристическим способом, однако существует ряд правил:

1. Если входное условие описывает область значений, например «Целое число принимает значение от 0 до 999», то существует один правильный класс эквивалентности и два неправильных.
2. Если входное условие описывает число значений, например «Число строк во входном файле лежит в интервале (1..6)», то также существует один правильный класс и два неправильных.
3. Если входное условие описывает множество входных значений, то определяется количество правильных классов, равное количеству элементов в множестве входных значений. Если входное условие описывает ситуацию «должно быть», например «Первый символ должен быть заглавным», тогда один класс правильный и один неправильный.
4. Если есть основание считать, что элементы внутри одного класса эквивалентности могут программой трактоваться по-разному, необходимо разбить данный класс на подклассы. На этом шаге тестирующий на основе таблицы должен составить тесты, покрывающие собой все правильные и неправильные классы эквивалентности. При этом составитель должен минимизировать общее число тестов.

Определение тестов:

1. Каждому классу эквивалентности присваивается уникальный номер.
2. Если еще остались не включенные в тесты правильные классы, то пишутся тесты, которые покрывают максимально возможное количество классов.
3. Если остались не включенные в тесты неправильные классы, то пишут тесты, которые покрывают только один класс.

#### **Анализ граничных значений**

**Граничные условия** — это ситуации, возникающие на высших и нижних границах входных классов эквивалентности.

**Анализ граничных значений отличается от эквивалентного разбиения следующим:**

1. Выбор любого элемента в классе эквивалентности в качестве представительного осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.
2. При разработке тестов рассматриваются не только входные значения (пространство входов), но и выходные (пространство выходов).

Метод требует определённой степени творчества и специализации в рассматриваемой задаче.

**Существует несколько правил:**

1. Построить тесты с неправильными входными данными для ситуации незначительного выхода за границы области значений. Если входные значения должны быть в интервале [-1.0 .. +1.0], проверяем −1.0, 1.0, −1.000001, 1.000001.
2. Обязательно писать тесты для минимальной и максимальной границы диапазона.
3. Использовать первые два правила для каждого из входных значений (использовать пункт 2 для всех выходных значений).
4. Если вход и выход программы представляет упорядоченное множество, сосредоточить внимание на первом и последнем элементах списка.

Анализ граничных значений, если он применён правильно, позволяет обнаружить большое число ошибок. Однако определение этих границ для каждой задачи может являться отдельной трудной задачей. Также этот метод не проверяет комбинации входных значений.

#### **Анализ причинно-следственных связей**

**Этапы построения теста:**

1. Спецификация разбивается на рабочие участки.
2. В спецификации определяются множество причин и следствий. Под причиной понимается отдельное входное условие или класс эквивалентности. Следствие представляет собой выходное условие или преобразование системы. Здесь каждой причине и следствию присваивается номер.
3. На основе анализа семантического (смыслового) содержания спецификации строится таблица истинности, в которой последовательно перебираются всевозможные комбинации причин и определяются следствия для каждой комбинации причин.

Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации, которые невозможны. Недостатком этого подхода является плохое исследование граничных условий.

#### **Предположение об ошибке**

Тестировщик с большим опытом выискивает ошибки без всяких методов, но при этом он подсознательно использует метод предположения об ошибке. Данный метод в значительной степени основан на интуиции. Основная идея метода состоит в том, чтобы составить список, который перечисляет возможные ошибки и ситуации, в которых эти ошибки могли проявиться. Потом на основе списка составляются тесты

**52. Тестирование ПО. Методы «Белого ящика»**

Белый ящик - то, содержимое чего известно.

*«Белый ящик»* — тестирование кода на предмет логики работы программы и корректности её работы с точки зрения компилятора того языка, на котором она писалась.

Техника тестирования по принципу Белого ящика, также называемая *техникой тестирования, управляемой логикой программы,* позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии тестировщик получает тестовые данные путеманализа логики работы программы.

Методы белого ящика основаны на знаниях о внутреннем устройстве проги, т.е. используя этот метод, тестируются алгоритмы, система управления кодом, сам код и др. внутренние механизмы.

Методы белого ящика начинают применять, когда приложение еще не закончено, т.е. в процессе разработки ПО.

Методы белого ящика направлены на локализацию ошибок, которые сложнее выявить, найти и зафиксировать. С их помощью можно *обнаружить логические ошибки и проверить степень покрытия тестами*.

Тестирование посредством белого ящика, как правило, включает в себя стратегию *модульного тестирования*, при котором тестирование ведется на модульном или функциональном уровне и работы по тестированию направлены на исследование внутреннего устройства модуля. На этом уровне тестирования проверяется управляющая логика, проявляющаяся на модульном уровне. Тестовые драйверы используются для того, чтобы все пути в данном модуле были проверены хотя бы один раз, все логические решения рассмотрены во всевозможных условиях, циклы были выполнены с использованием верхних и нижних границ и проконтролированы структуры внутренних данных.

Методы тестирования на основе стратегии белого ящика:

Ввод неверных значений. При вводе неверных значений тестировщик заставляет коды возврата показывать ошибки и смотрит на реакцию кода. Это хороший способ моделирования определенных событий, например переполнения диска, нехватки памяти и т.д.

Модульное тестирование. При создании кода каждого модуля программного продукта проводится модульное тестирование для проверки того, что код работает верно и корректно реализует архитектуру. При модульном тестировании новый код проверяется на соответствие подробному описанию архитектуры; обследуются пути в коде, устанавливается, что экраны, ниспадающие меню и сообщения должным образом отформатированы; проверяются диапазон и тип вводимых данных, а также то, что каждый блок кода, когда нужно, генерирует исключения и возвращает ошибки (еггог returns). Тестирование каждого модуля программного продукта проводится для того, чтобы проверить корректность алгоритмов и логики и то, что программный модуль удовлетворяет предъявляемым требованиям и обеспечивает необходимую функциональность. По итогам модульного тестирования фиксируются ошибки, относящиеся к логике программы, перегрузке и выходу из диапазона, времени работы и утечке памяти.

Тестирование обработки ошибок. При использовании этого метода признается, что нереально на практике проверить каждое возможное условие возникновения ошибки. По этой причине программа обработки ошибок может сгладить последствия при возникновении неожиданных ошибок. Тестировщик обязан убедиться в том, что приложение должным образом выдает сообщение об ошибке. Так, приложение, которое сообщает о системной ошибке, возникшей из-за промежуточного программного обеспечения представляет небольшую ценность, как для конечного пользователя, так и для тестировщика.

Утечка памяти. При тестировании утечки памяти приложение исследуется с целью обнаружения ситуаций, при которых приложение не освобождает выделенную память, в результате чего снижается производительность или возникает тупиковая ситуация.

Комплексное тестирование. Целью комплексного тестирования является проверка того, что каждый модуль программного продукта корректно согласуется с остальными модулями продукта. При комплексном тестировании может использоваться технология обработки сверху вниз и снизу вверх, при которой каждый модуль, являющийся листом в дереве системы, интегрируется со следующим модулем более низкого или более высокого уровня, пока не будет создано дерево программного продукта. Эта технология тестирования направлена на проверку не только тех параметров, которые передаются между двумя компонентами, но и на проверку глобальных параметров и, в случае объектно-ориентированного приложения, всех классов верхнего уровня.

Тестирование цепочек. Тестирование цепочек подразумевает проверку группы модулей, составляющих функцию программного продукта. Эти действия известны еще как модульное тестирование, с его помощью обеспечивается адекватное тестирование компонентов системы. Данное тестирование выявляет, достаточно ли надежно работают модули для того, чтобы образовать единый модуль, и выдает ли модуль программного продукта точные и согласующиеся результаты.

Исследование покрытия. При выборе инструмента для исследования покрытия важно, чтобы группа тестирования проанализировала тип покрытия, необходимый для приложения. Исследование покрытия можно провести с помощью различных технологий. Метод покрытия операторов часто называют С1, что также означает покрытие узлов. Эти измерения показывают, был ли проверен каждый исполняемый оператор. Данный метод тестирования обычно использует программу протоколирования (profiler) производительности.

Покрытие решений. Метод покрытия решений направлен на определение (в процентном соотношении) всех возможных исходов решений, которые были проверены с помощью комплекта тестовых процедур. Метод покрытия решений иногда относят к покрытию ветвей и называют С2. Он требует: чтобы каждая точка входа и выхода в программе была достигнута хотя бы единожды, чтобы все возможные условия для решений в программе были проверены не менее одного раза и чтобы каждое решение в программе хотя бы единожды было протестировано при использовании всех возможных исходов.

Покрытие условий. Покрытие условий похоже на покрытие решений. Оно направлено на проверку точности истинных или ложных результатов каждого логического выражения. Этот метод включает в себя тесты, которые проверяют выражения независимо друг от друга. Результаты этих проверок аналогичны тем, что получают при применении метода покрытия решений, за исключением того, что метод покрытия решений более чувствителен к управляющей логике программы.

**51. Критерии качества ПО**

**Качество программного обеспечения** — способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям; весь объем признаков и характеристик программ, который относится к их способности удовлетворять установленным или предполагаемым потребностям.

ГОСТ 28195-89 содержит следующий список комплексных показателей качества программного продукта:

*1) надежность* – это его способность с достаточно большой вероятностью безотказно выполнять определенные функции при заданных условиях и в течение заданного периода времени;

*2) корректность*

*3) удобство применения -* эргономичность программы, легкость, с которой пользователь может использовать программу для своих целей;

*4) сопровождаемость*

*5) эффективность* – соотношение уровня услуг, предоставляемых ПО пользователю при заданных условиях, и объема используемых для этого ресурсов. К числу таких ресурсов могут относиться требуемые аппаратные средства, время выполнения программ, затраты на подготовку данных и интерпретацию результатов;

*6) универсальность.*

В международном стандарте ISO 9126 прописаны следующие показатели:

*1) эффективность -* количество ресурсов системы, потребляет программа (время процессора, размер памяти, внешняя память, ширина канала сети, а также взаимодействия с пользователем). Чем меньше ресурсов потребляется, тем лучше;

*2) надежность -* это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под отказом в ПС понимают проявление в нем ошибки. Надежное ПС не исключает наличия в нем ошибок - важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко.

*3) сопровождаемость -* это характеристики ПО, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

*4) функциональность -* это способность ПО выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПО.

*5) практичность*

*6) мобильность -* это способность ПС быть перенесенным из одной среды (окружения) вдругую, в частности, с одного компьютера на другой.

Комплексные показатели называют также факторами качества или характеристиками качества.

Каждому из комплексных показателей соответствует определенный набор критериев качества. В свою очередь, каждый из критериев определяется своими метриками, которые составляются из оценочных элементов, определяющих заданное в метрике свойство. На разных этапах ЖЦ для разных классов программных продуктов применяются разные критерии качества.

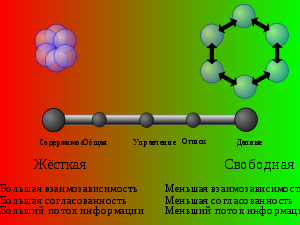
Показатели качества программного продукта можно разделить на две категории: *внутренние* и *внешние* . Внешние факторы качества (корректность, надежность, эффективность, удобство использования) могут быть определены, исходя из поведения программного продукта, и являются видимыми пользователю. В отличие от них внутренние показатели качества (переносимость, понятность, модифицируемость, сопровождаемость) для пользователя не видны и не важны. Однако эти показатели чрезвычайно важны для разработчиков, поскольку в конечном итоге именно они определяют экономические затраты на разработку требуемого программного продукта. Внутренние показатели более важны для проекта, чем внешние, поскольку внешние характеристики практически всегда от них зависят.

**50.Связанность и цельность модулей**

**Связанность** (или **зависимость)** — характеристика взаимосвязи модуля с другими модулями. Это степень, в которой каждый программный модуль полагается на другие модули.

Связанность обычно противопоставляется [связности](http://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) (Связность — характеристика *внутренней* взаимосвязи между частями *одного модуля. Иное название связности -* ***цельность***). Слабая связанность часто сочетается с сильной связностью и наоборот. [Метрика качества ПО](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D1%80%D0%B8%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) связанности и связности была придумана [Ларри Константином](http://en.wikipedia.org/wiki/Larry_Constantine), изначальным разработчиком Структурного Дизайна, который был также ранним сторонником таких концепций (см. также [SSADM](http://en.wikipedia.org/wiki/Structured_Systems_Analysis_and_Design_Methodology)). Слабая связанность часто является признаком хорошо структурированной компьютерной системы и признаком хорошего дизайна, и, когда она комбинируется с сильной связностью, соответствует общим показателям хорошей читаемости и поддерживаемости.

## Типы связанности



*Концептуальная модель связанности*

Связанность может быть «низкой» (также «свободной» и «слабой») или «высокой» (также «несвободной» или «сильной»). Некоторые типы связанности по направлению от наивысшей к наинизшей связанности, следующие:

Связанность содержимого (высокая)

Связанность содержимого это когда один модуль изменяет или полагается на внутреннюю кухню другого модуля (например, доступ к локальным данным другого модуля).

Следовательно, изменение способа, которым второй модуль производит данные (расположение, тип, время) приведёт к необходимости изменять зависимый модуль.

Общая связанность

Общая связанность это когда два модуля делят между собой один и тот же глобальный набор данных (например, глобальную переменную). Изменение общего ресурса подразумевает изменение всех модулей, использующих его.

Внешняя связанность

Внешняя связанность возникает когда два модуля делят между собой определённый извне формат данных, протокол связи, или интерфейс устройства. Это обычно основано на коммуникации со внешними средствами и устройствами.

Связанность управления

Связанность управления это когда один модуль управляет исполнением другого путём передачи ему информации по поводу того, что делать (например, передачей флага «что-нужно-сделать»).

Связанность по отпечатку в структуре данных (stamp coupling)

Это когда модули делят между собой составную структуру данных, и каждый используют только её часть, по возможности даже не ту же самую часть (например, передача полной структуры в функцию, которой нужно только одно поле этой структуры). Это может привести к изменению способа чтения модулем этой структуры, когда изменится поле, которое модуль не использует.

Связанность данных

Связанность данных это когда модули делят общие данные через, скажем, параметры. Каждая порция данных это элементарный фрагмент, и это только те данные, которые используются совместно (например, передача целого числа функции, вычисляющей квадратный корень).

Связанность сообщений (низкая)

Это наислабейший тип связанности. Он может быть достигнут децентрализацией состояний (как в объектах) и коммуникацией компонентов через параметры или передачу сообщений.

Нет связанности

Модули не общаются между собой вовсе.

### *Связанность В ООП:*

Связанность подклассов

Описывает взаимоотношения между дочерним и родительским элементами. Дочерний элемент связан с родительским, но родительский не связан с дочерним.

Временнáя связанность

Когда два действия связаны вместе в один модуль только потому, что они происходят в одно и то же время.