**НПО**

### **Основные понятия надежности аппаратного обеспечения компьютерных систем надежность, безотказность, отказ, наработка до отказа, сбой, работоспособное состояние, устойчивость, достоверность).**

Основные понятия и определения в области надежности аппаратного обеспечения КС содержит межгосударственный стандарт **ГОСТ27.002-89 «Надежность в технике. Основные понятия. Термины и определения»**. Стандарт переиздан в ноябре 2008.

***Надежность*** – свойство технического О сохранять во времени в установленных пределах значения всех параметров, характеризующих способность выполнять требуемые функции в заданных режимах и условиях применения, технического обслуживания, хранения и транспортирования.

***Объект*** - компьютерная система (ex., компьютерная сеть), либо её подсистема (ex., компьютер, винчестер, принтер). Очевидно, что в течение эксплуатации О расходует свои ресурсы и в конце концов прекращает свою эксплуатацию. Надежность является сложным свойством, которое в зависимости от назначения О и условий его применения состоит из сочетания свойств.

– безотказности;

– ремонтопригодности (восстанавливаемости);

– долговечности;

– сохраняемости.

***Безотказность*** – свойство О непрерывно сохранять работоспособное состояние в течение некоторого времени или наработки.(без сбоев)

***Ремонтопригодность*** – свойство О, заключающееся в приспособленности его к поддержанию и восстановлению работоспособного состояния путём технического обслуживания и ремонта.

***Долговечность***– свойство О сохранять работоспособное состояние до наступления предельного состояния, при установленной системе путём технического обслуживания и ремонта.(со сбоями)

***Сохраняемость*** – свойство О сохранять в заданных пределах значения параметров, характеризующих способность О выполнять требуемые функции, в течение и после хранения и (транспортирования).

Понятие надёжности является фундаментальным понятием, которое охватывает все стороны технической эксплуатации систем.

***Наработка*** – продолжительность или объем работы О. Выражается чаще всего через t, может быть в другой форме: количество циклов работы, количество решённых задач и т.д.

***Отказ*** – событие, заключающееся в нарушении работоспособного состояния О. Отказ почти всегда вызывается физическим разрушением объекта (полностью или частично).

Примеры отказов. Для транзистора отказ возникает часто из-за пробоя эмиттерного или коллекторного переходов. В результате такого отказа две точки оказываются соединёнными накоротко. Это внезапный отказ типа «короткое замыкание», аналогично и для конденсатора.

У транзисторов существует и другой тип отказа, когда происходит сгорание внутренних соединений кристалла с внешними контактами, или нарушение внешних контактов. В этом случае сопротивление между двумя точками равно бесконечности. Такой отказ носит название внезапного отказа типа «обрыв», аналогично и у резисторов.

***Наработка до отказа*** – наработка О от начала эксплуатации до возникновения **1-го** отказа.

***Сбой*** – самоустраняющийся отказ или однократный отказ, устраняемый незначительным вмешательством оператора. Чаще всего для компьютерных систем сбои вызываются внутренними или внешними помехами, имеющими электромагнитный характер. Таким образом, сбой – кратковременное нарушение правильной работы объекта, после которого работоспособность объекта самовосстанавливается или восстанавливается оператором без проведения ремонта.

***Работоспособное состояние*** – это состояние объекта, при котором значение всех параметров, характеризующих способность выполнять заданные функции, соответствуют требованиям нормативно – технической и (или) конструкторской (проектной) документации.

Основные качества компьютеров и КС, связанные с надежностью, определяются распределением отказов во времени, процессами восстановления и организацией обслуживания.

Улучшение надёжности требует дополнительных затрат при разработке, изготовлении и эксплуатации. В общем случае, установление требований к надёжности компьютеров и КС представляет собой сложную технико-экономическую задачу, решаемую по критерию минимума суммарных расходов.

В случае, когда отказ компьютера и КС может повлечь за собой опасность для жизни людей, уровень надёжности выбирается из требования, чтобы вероятность отказа соответствовала практически невозможному событию.

Подводя итог вышеизложенного, можно сказать что надёжность – сложное комплексное свойство объекта, которая может разлагаться на более простые (частные) свойства. Рассмотренные нами 4 свойства надёжности рекомендуется для широкого класса объектов. Однако для КС этих свойств надёжности оказалось недостаточно. Поэтому находят применение дополнительные свойства.

***Устойчивость (живучесть)*** – свойство объекта сохранять работоспособность полностью или частично в условиях неблагоприятных воздействий, не предусмотренных нормальными условиями эксплуатации.

Главный смысл требований к живучести О: он длительное время работал непрерывно без отказа в нормальных условиях эксплуатации, и чтобы его можно было быстро отремонтировать, но также чтобы объект в ненормальных условиях эксплуатации сохранял работоспособность, хотя бы и ограниченную.

***Достоверность информации выдаваемой объектом*** – свойство объекта производить безошибочно преобразование, хранение и передачу информации. Показателем достоверности обычно является вероятность искажения либо потери информации в одном знаке.

### **Показатели надежности невосстанавливаемых объектов ( P(t), Q(t), f(t)).**

Вероятность безотказной работы ***P(t)*** выражает вероятность того, что невосстанавливаемый объект не откажет к моменту времени ***t***, т.е. в течение заданной наработки ***(0, t).***

***P(t) = P { T>= t },***

где ***T*** – случайная величина наработки до отказа.

Свойства показателя ***P(t)***:

1) ***P(0) =1***,

т.е. до начала работы объект работоспособен;

2)  ***,***

т.е. предполагается, что объект не может сохранить свою работоспособность неограниченно долго;

3) ,

т.е. предполагается, что объект не может после отказа спонтанно восстанавливаться (для восстанавливаемых объектов этот показатель не используется).

В качестве показателя надёжности неудобно использовать функциональную зависимость ***P(t)*.**Поэтому в технических условиях обычно оговаривают отдельные ординаты функции ***P(t)*** при значениях ***t***, выбираемых из нормированного ряда: ***t*** =100; 500; 1000; 2000; 5000; 10000.

Иногда ***P(t)*** называют функцией надёжности.

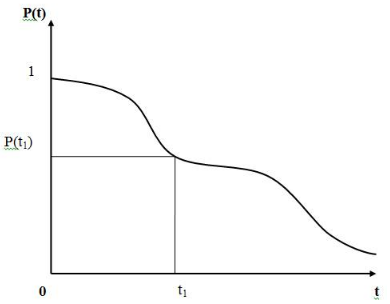


Рис. 1. Зависимость вероятности безотказной работы от времени

***Вероятность отказа объекта*** – вероятность того, что О откажет хотя бы один раз в течение заданной наработки, будучи работоспособным в начальный момент времени. Вероятность отказа в течение заданной наработки ***(0, t).***

***Q(t) = 1 – P(t) = P{T < t },***

где: ***Q(t)*** – вероятность того, что наработка до отказа ***Т*** меньше заданной наработки ***t*.** Фактически ***Q(t)***представляет собой функцию распределения СВ ***Т***.

На рис. 2.2 представлены функции распределения случайной величины ***T*** ***Q(t)*** и плотности распределения наработки на отказ ***f(t)***. Таким образом, ***Q(t)*** – функция распределения времени наработки на отказ.

Из теории вероятности:

, **(2.1)**

где: ***ft (x)*** – функция плотности распределения времени до отказа; ***х***–переменная интегрирования.

***F(t)=P(T<t).***

Тогда

.

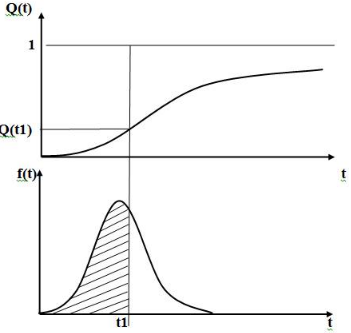


Рис. Функции распределения случайной величины и плотности распределения наработки до отказа

Статистическое определениеопределяется по статистическому эксперименту. ,

где ***N(t)*** – количество объектов безотказно работающих до момента времени t;

***N0***– исходное количество объектов.

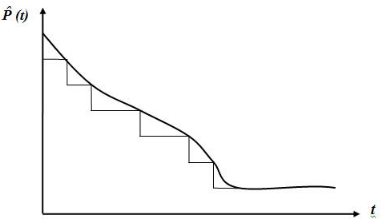
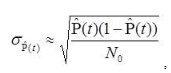


Рис. Статистическое определение

Точки разрывасоответствуют отдельным отказам. Видот эксперимента к эксперименту изменяется случайным образом.

Среднеквадратическая погрешность оценкиопределяется на основе теоремы Муавра-Лапласа:

**(2.2)**

где– значениедля рассматриваемого момента времени **t**.

Из формулы **(2.2)** следует, что для обеспечения малой погрешности оценки требуется эксперимент над большим числом объектов.

Чтобы уменьшить ошибку в 10 раз, т.е. получить в ответе верный десятичный знак, нужно увеличить N0 – число объектов в 100 раз.

Статистическое определение:



где **n(t)**- количество объектов, отказавших к моменту времени **t***.*

Можно по другому:



Чтобы отличать в дальнейшем от функции надежности вероятность безотказной работы в течение фиксированной наработки (заданной) в последнем случае буква ***t*** имеет индекс, т.е. ***ti***, а значит ***P(t) –*** функция надёжности, а ***P(ti )*** – вероятность безотказной работы в течение заданной наработки в технике ***(0, ti)***.

***Плотность распределения наработки до отказа*** – есть производная от функции ненадёжности.

Из формулы (2.1) следует:



Произведение ***f(t)dt*** представляет собой безусловную вероятность того, что объект откажет на интервале времени ***(t, t+dt)****.*

Напомним, что любая функция ***f(x)*** является функцией плотности, если обладает следующими свойствами:

Рассмотрим статистическое определение ⇁*f(t)*.



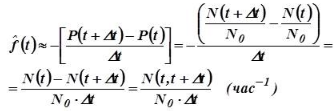
Обозначим:

**N(t)**– число объектов, не отказавших к моменту времени ***t***;

– число объектов, не отказавших к моменту времени.

-достаточно малый промежуток времени.

Тогда из **(2.3)** следует:



– число отказавших объектов на интервале;

**N0**– первоначальное число объектов, поставленных на испытания.

Таким образом, значение функцииравно числу отказавших объектов, отнесенное к общему числу объектов  **N0**, поставленных на испытания за период.

Фактически ***f(t)*** представляет собой безусловную плотность вероятности возникновения отказа.

### **Показатели надежности невосстанавливаемых объектов (λ(t), mt, Dt, σt).**

***Интенсивность отказов*** ***λ(t)*** – условная плотность вероятности возникновения отказа объекта, определяемая при условии, что до рассматриваемого момента времени отказ не возник:

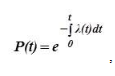
.

Так как:

то

**(2.4)**

Решая (2.4), получим связь между *λ(t)* и *P(t)*:



где ***P(t)*** – вероятность безотказной работы на отрезке ***(0, t).***

При ***λ = const.***



При



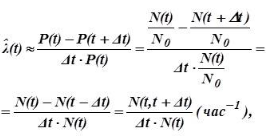
Рассмотрим ***статистическое определение λ(t).***

Обозначим:

***N(t)* –** число объектов, не отказавших к моменту ***t***;

***N(t+∆t)*** *-*число объектов, не отказавших к моменту***t+∆t*** (***∆t* –** достаточно малый интервал времени).

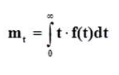
Тогда



где ***N(t,t+∆t)***– число отказавших объектов на интервале ***(t,t+∆t)***.

Таким образом, равно числу отказавших объектов за интервал времени ***∆t***, отнесённое к числу неотказавших объектов к моменту времени ***t*** за период ***∆t***.

Средняя наработка до отказа определяется как математическое ожидание времени до отказа (фактически это время до первого отказа).

 **(2.5)**

Интегрируя по частям выражение (2.5), получим



т.е. средняя наработка численно равна площади под кривой ***P(t)****.*

При ***λ = const.***



где mt - самый естественный показатель надёжности, но он ничего не говорит о характере распределения времени до отказа.

Например, две совершенно различные функции P1(t) и P2(t), выражающие резко отличающиеся вероятности безотказной работы (рис. 2.5), могут характеризоваться одинаковыми значениями средней наработки до отказа ***mt1=mt2***. При этом заштрихованные области на рис.2.5 равны между собой, то есть ***S1=S2***.

Чтобы различать такие случаи, наряду с mt применяется показатель – среднеквадратическое отклонение наработки до отказа.



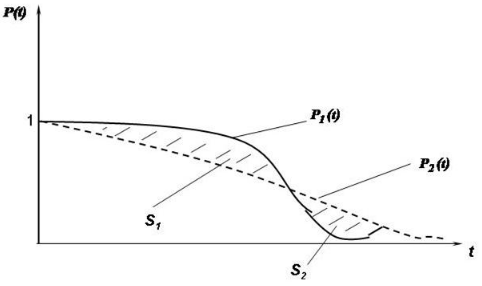
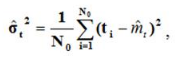


Рис. Виды функций безотказной работы(

где ***Dt*** – дисперсия, характеризующая величину разброса наработки относительно среднего значения.

Статистические оценки mt и :





где ***ti*** – время до отказа ***i***-ого объекта.

Оценки  – приведены для случая, когда эксперимент проводился до отказа всех объектов.

### **Показатели надежности восстанавливаемых объектов (Kr(t), Kr ст.). Экспоненциальный закон распределения времени до отказа.**

***Коэффициент или функция готовности KГ(t)*** – вероятность того, что объект окажется в работоспособном состоянии в произвольный момент времени, кроме планируемых периодов, в течение которых применение объекта по назначению не предусматривается .

Статистически:



где:

***N0*** –***NB(t)*** **–** количество объектов, находящихся в момент **t** в состоянии восстановления (ремонта);

***NB(t)*** **–** число объектов, находящихся в рабочем состоянии в момент времени ***t;***

***N0*–** общее число объектов.

**Коэффициент готовности *KГ(t)* используется в том случае, когда кроме факта отказа необходимо учитывать и время восстановления объекта.**

***Стационарный коэффициент готовности*** определяется как



Другой способ определения стационарного коэффициента готовности:



где: ***S tp*** – суммарное время работы объекта;

***S tв***– суммарное время восстановления объекта.

Рис. 2.6 иллюстрирует зависимость коэффициента готовности от времени.

Следует отметить, что ***КГ(t)*** и ***КГ.СТ***. — комплексные показатели, учитывающие свойство безотказности и восстанавливаемости.



Рис. Зависимость коэффициента готовности от времени

Время между соседними отказами для аппаратуры компьютера является непрерывной СВ, которая характеризуется некоторым законом распределения. Зависимость надёжности от времени описывается с помощью математической модели надёжности (ММН), то есть математического выражения (формулы, уравнения, система уравнений, алгоритма), позволяющего определять показатели надёжности ММН в виде формул с эмпирическими коэффициентами. ММН в виде формул с эмпирическими коэффициентами носят название статистических моделей распределения. Наиболее распространённой статистической моделью надёжности является ***экспоненциальная модель*** распределения времени до отказа, по которой вероятность безотказной работы определяется выражением:



где ***λ*** — параметр модели.

Функция плотности:



Интенсивность отказов:



Из практики эксплуатации радиоэлектронных приборов график типичной зависимости интенсивности отказов от **t** имеет вид, представленный на рис. 2.8.

На рис. 2.8 через ***I*** обозначен период приработки, ***II*** *–* период эксплуатации, ***III*** *–* период интенсивного старения.

Период приработки ***I*** имеет повышенную интенсивность отказов, вызванную приработочными отказами, обусловленными дефектами производства, монтажа и наладки.

Таким образом, экспоненциальная модель момент быть использована в случае, когда





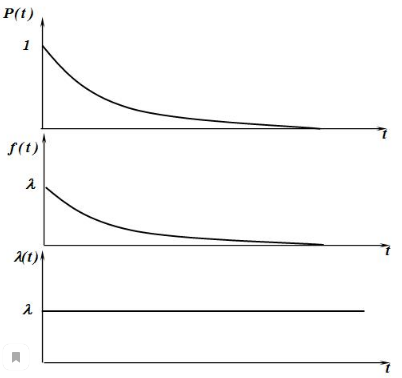


Рис. Экспоненциальная модель надежности

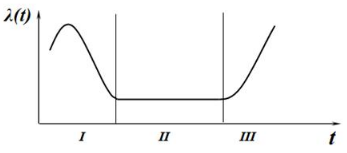


Рис. 2.8. Зависимость интенсивности отказов от времени ***t***

### **Общая схема расчета надежности компьютерных систем. Расчет надежности невосстанавливаемых резервируемых и нерезервируемых компьютерных систем. Пример.**

Для расчета надежности необходимо иметь модель надежности системы, которая составляется на основе функциональной схемы системы. Предполагается, что отказы элементов независимы, элементы и система могут находиться в работоспособном или неработоспособном состояниях.

При составлении модели надежности системы функциональные связи между элементами системы заменяются логическими, характеризующими безотказную работу системы в зависимости от работоспособности или неработоспособности элементов.

Элемент, при отказе которого отказывает вся система, считается последовательно соединенным в модели надежности. В этом случае безотказная работа системы имеет место при сохранении работоспособности всеми элементами последовательного соединения. Отказ системы наступает при отказе хотя бы одного элемента.

Элемент, отказ которого не приводит к отказу всей системы, считается включенным параллельно в модели надежности. Безотказная работа всей системы при параллельном соединении элементов имеет место при сохранении работоспособности хотя бы одного элемента.

Способ вычисления показателей надежности существенно зависит от типа соединения отдельных конструктивных элементов и их взаимодействия между собой с точки зрения обеспечения безотказной работы системы.

*1) Нерезервированные системы:*

Рассмотрим систему, состоящую из n элементов (подсистем, рис. 2.9). При этом отказ всей системы происходит при отказе любого элемента. Такая система является нерезервированной.



Рис. Модель надежности последовательного соединения элементов

Надежность такой системы определяется по теореме умножения вероятностей для независимых событий и равна произведению вероятностей безотказной работы отдельных элементов Pi(tj) в течение наработки (0, tj).

Тогда:



где **n** – количество последовательно соединенных элементов; **Pi(tj)** – вероятность безотказной работы i-го последовательно соединённого элемента.

*2) Резервированные системы*

Резервированная система изображена на рис. 2.10.

Отказ такой системы происходит только в том случае, когда откажут все элементы системы.

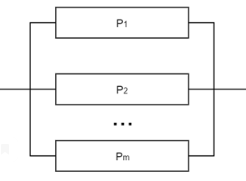


Рис. 2.10. Модель надежности параллельного соединения m элементов.

Вероятность безотказной работы резервированной системы, представленной на рис. 2.10, равна:



где **Pi(tj)** – вероятность безотказной работы i-го параллельно соединенного элемента системы; **m** – количество параллельно соединенных элементов.

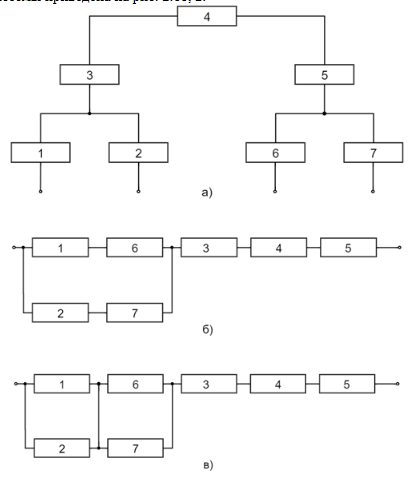


Рис. 2.11. Схемы системы связи с использованием ИСЗ: а – схема системы с ИСЗ; б – модель надежности при частотном разделении каналов;

в – модель надежности при временном разделении каналов; 1, 2- передатчики телевизионных программ; 3, 5 – антенные устройства; 4 – ИСЗ;

6, 7 - приемники телевизионных программ.

В качестве показателя надежности принимается вероятность того, что хотя бы один телевизионный канал исправлен в течение наработки (0, tj). Разделение каналов может осуществляться двумя способами: а) частотным, при котором каждый передатчик закреплен за определенным приемником; б) временным, при котором каждый передатчик может работать с любым приемником.

Исходными данными являются вероятности безотказной работы в течение наработки tj всех подсистем Pi(tj), где i=1, 2, ..., 7.

При частотном разделении каналов каждый передатчик закреплен за определенным приемником, поэтому отказ передатчика любого канала приводит к невозможности использования соответствующего приемника. Для того чтобы хотя бы один телевизионный канал был работоспособен, необходима безотказная работа устройств например, 1 и 6 или 2 и 7, а также 3, 4 и 5. Следовательно, модель надежности системы имеет вид, представленный на рис. 2.11, б.

В соответствии с данной моделью надежности вероятность безотказной работы системы при частотном разделении каналов Pчаст имеет вид :

Pчаст (tj) = {1 – [1 – P1(tj) ∙ P6(tj)] ∙ [1 – P2(tj) ∙ P7(tj)]} ∙ P3(tj) ∙ P4(tj) ∙ P5(tj).

При временном разделении каналов каждый передатчик может работать с любым приемником. Поэтому для безотказной работы системы на интервале (0, tj) необходимо, чтобы были исправны хотя бы один передатчик и хотя бы один приемник. Модель надежности данной системы приведена на рис. 2.11, в.

В соответствии с данной моделью надежности вероятность безотказной работы системы может быть определен по формуле

Pвр(tj) = {1 – [1 – P1(tj)] ∙ [1 – P2(tj)]} ∙ {1 – [1 – P6(tj)] ∙ [1 – P7(tj)]} ∙ P3(tj) ∙ P4(tj) ∙ P5(tj).

### **Связь эффективности и надежности компьютерных систем.**

**Эффективностью системы** называется свойство системы выдавать некоторый полезный результат (эффект) при ее использовании в соответствии с назначением.

При определении показателей эффективности систем рассматривают результат применения системы, который почти всегда имеет элементы случайности. Поэтому в качестве показателей эффективности часто применяют характеристики случайных событий, случайных величин или случайных процессов.

Если задачей применения системы является осуществление некоторого события, то за показатель эффективности принимают вероятность выполнения этого события.

Когда результат применения системы можно рассматривать как случайную величину, то за показатель эффективности системы принимают параметры закона распределения этой случайной величины (математическое ожидание, дисперсия и т.д.)

Большинство сложных систем являются восстанавливаемыми, и в процессе эксплуатации существует возможность восстанавливать отказавшие элементы без прекращения функционирования всей системы (например, глобальная сеть Интернет, локальная сеть и т.д.).

При отказах отдельных элементов сложной системы может уменьшаться эффективность функционирования всей системы.

В общем, эффективность системы зависит от множества факторов, в том числе от :

* надежности;
* производительности;
* защищенности;
* точности;
* стоимости;
* и т.д.

Надежность систем является одной из важнейших характеристик систем. Очевидно, что чем выше надежность системы, тем выше и её эффективность, но лишь до некоторого предела. Зависимость эффективности от надежности показана на рис. 2.12.

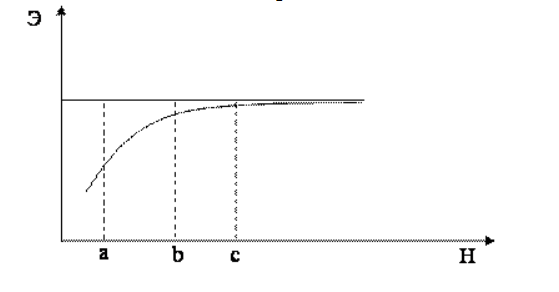


Рис. 2.12. Графическое представление связи между эффективностью и надежностью

Из рисунка видно, что на участке *a-b* изменение надежности существенно влияет на эффективность. Повышение же надежности выше уровня *c* нецелесообразно с точки зрения повышения эффективности.

Различают различные виды эффективности объектов (в том числе систем):

· *Эффективность номинальная (идеальная)*  – эффективность объекта при безотказном его состоянии.

· *Эффективность реальная* – эффективность реального объекта, не обладающего идеальной надежностью.

· *Эффективность техническая* – технический эффект, полученный при использовании объекта (количество переданной информации, снижение затрат времени и т.д.);

· *Эффективность экономическая* – степень выгодности экономических затрат при использовании объекта.

В дальнейшем под эффективностью системы мы будем понимать в первую очередь *техническую эффективность*.

Например, снижение затрат времени на прибытие машины скорой помощи к пострадавшему – показатель технической эффективности АСУ «Скорая помощь».

Вместе с тем, понятие эффективности системы нецелесообразно применять к малым элементам системы. Например, не рассматривают эффективность применения блока питания в персональном компьютере. Для отдельного объекта, имеющего определенное целевое назначение и самостоятельное применение (например, для компьютера), имеет смысл рассматривать как надежность, так и эффективность.

Для элементов системы (например, винчестер компьютера) целесообразно рассматривать только надежность, так как эти элементы не имеют самостоятельного применения.

Для очень сложных систем (например, сеть Интернет) имеет смысл только эффективность, так как при отказах отдельных элементов системы может лишь ухудшить качество функционирования всей системы.

В общем, для оценки надежности объекта могут использоваться различные показатели надежности. При этом необходимо учитывать тот факт, что основным показателем надежности должен быть выбран тот, который входит в формулу для определения эффективности.

Показатели эффективности могут носить, например, или вероятностный, или информационный, или стоимостный характер, также они могут быть безразмерными. Связь между показателями эффективности и надёжности системы может быть установлена различными способами. Рассмотрим один из них.

Показатели надежности системы или элементов системы учитываются при вычислении значений показателя эффективности [22]. В общем, удается отследить снижение эффективности из-за её недостаточной надежности, при этом вычисляется идеальное значение эффективности (**Эи**) при абсолютной надежности системы и некоторое «реальное» значение (**Эр**), учитывающее фактическую надежность системы.

Снижение эффективности из-за недостаточной надежности системы характеризуют следующие выражения:

**∆Э = Эи – Эр ;**

**∆Э / Эи ;**

**Эр / Эи .**

Например, в качестве показателя эффективности системы связи может быть принята величина годового дохода от эксплуатации каналов связи. Тогда при вычислении **Эи** определяется величина идеального дохода, который может быть получен при условии, что в системе не происходили бы отказы. А при вычислении **Эр** – величина реального дохода при возможных отказах в системе.

При оценке эффективности целесообразно выделять дискретные состояния, в которых может находиться система. Тогда значение показателя эффективности вычисляется по формуле, схожей с формулой математического ожидания:

**Э = ∑(Эj ∙ Рj) ,j ∈ G**

где:

**G** – множество состояний системы;

**Эj** – условный показатель эффективности при нахождении системы в j-ом состоянии;

**Pj** – вероятность пребывания системы в j-ом состоянии.

В общем, выбор размерности показателя эффективности зависит от назначения системы и выполняемых ею функций. Например, для системы связи размерность показателя эффективности может быть выражена в единицах информации, компьютерных систем в единицах времени и т.д. Почти всегда для всех систем эффективность системы может быть выражена в единицах стоимости.

2.5.4. Пример расчета эффективности систем

Система связи состоит из двух подсистем и предназначена для передачи данных по N = 200 одинаковым каналам связи. 1-ая подсистема передаёт данные по N1 = 120 каналам, 2-ая – по N2 = 80 каналам. Вероятность безотказной работы 1-й подсистемы P1(tj) = 0,95, второй – P2(tj) = 0.97. Доход, получаемый от эксплуатации каждого канала в течение tj = 10 часов, равен двум условным единицам (у.е). В случае выхода канала из строя доход теряется. Определить среднюю величину потери дохода [22].

Величина идеального дохода равна:

**Эи= N ∙ 2 = 400 у.е.**

Возможны четыре состояния системы.

Состояние 1.

Обе подсистемы работоспособны.

**Э1 = N ∙ 2 = 400 у.е.;**

**PI = P1(tj) ∙ P2(tj) =0,9215.**

Состояние 2.

1-ая подсистема работоспособна, 2-ая отказала.

**Э2 = (N – N2) ∙ 2 = N1 ∙ 2 = 240 у.е.;**

**PII = P1(tj) ∙ (1- P2(tj)) = 0,0285.**

Состояние 3.

2-ая подсистема работоспособна, 1-ая отказала.

**Э3 = (N – N1) ∙ 2 = N2 ∙ 2 = 160 у.е.;**

**PIII = P2(tj) ∙ (1- P1(tj)) = 0,0485.**

Состояние 4.

Обе подсистемы отказали.

**Э4 = (N – N1 – N2) ∙ 2 =0;**

**PIV = (1- P1(tj)) ∙ (1- P2(tj)) = 0,05 ∙ 0,03 = 0,0015.**

Отсюда следует, что величина реального дохода для **j = 1..4** равна

**Эр = ∑ (Эj ∙ Pj) = 400 ∙ 0,9215 + 240 ∙ 0,0285 + 160 ∙ 0,0485 = 383,2 у.е.;**

**∆Э = Эид – Эр = 400 – 383,2 = 16,8 у.е.**

Снижение эффективности функционирования системы произошло на 16,8 у.е. вследствие ее недостаточной надежности.

### **Понятие требований к системам и ПО. Характеристики для оценки отдельных требований по ISO/IEC/IEEE 29148:2011.**

Очевидно, что разработчик может создать качественное ПО, если понимает реальные потребности пользователя.

В общем потребности пользователей не конкретны и не однозначны, и обычно пользователь описывает свои пожелания в абстрактной форме, например:

1. Я хочу, чтобы разрабатываемое ПС обеспечивало дружественный интерфейс.

2. Я хочу, чтобы разрабатываемое ПС предоставляло текущую информацию о клиентах банка.

Таким образом, обычно пользователь описывает свойства программного обеспечения, которые ему необходимы. На данном уровне детализации требования к ПС обычно называются *пользовательскими требованиями* или *функциями* (от английского feature – свойства, функции), хотя в данном контексте это могут быть как функциональные свойства ПС, так и нефункциональные. Функции представляют собой описание желаемого поведения разрабатываемого ПС и служат основой для формирования *требований разработчика* к ПС. В общем случае каждая пользовательская функция порождает одно или несколько требований разработчика.

Требования разработчика представляют собой детализированные описания пользовательских функций. Например, требование пользователя имеет следующий вид: «Разрабатываемое ПС должно предоставлять текущую информацию о вкладах клиентов банка». Очевидно, что данная функция не является требованием разработчика, потому что здесь ничего не говорится о формате выводимых данных, о количестве атрибутов, о типе валют и т.д. Такую пользовательскую функцию можно представить набором требований разработчика. Например, к одному из таких требований может относиться следующее:

«Информацию о текущих вкладах предоставлять в виде печатной формы, представленной в виде таблицы».

Документом, в котором представлены требования к разрабатываемому программному средству, является *спецификация требований к программному обеспечению (СТПО)*.

***Спецификация*** представляет собой формализованное представление требований, предъявляемых к программе, которые должны быть удовлетворены при ее разработке, а также описание условий и эффекта действий без указания способа их достижения.

Спецификация требований к ПОявляется техническим документом, в котором описаны *условия функционирования* для отдельного программного продукта или набора программ, которые выполняют определенные задачи в определенной среде. Спецификация требований к ПОиграет важную роль в общем плане проекта, так как она может описывать не только отдельный программный продукт, но также тот, который является частью большей системы. В случае, когда спецификация требований к ПОявляется частью большей системы, у разработчика должна иметься спецификация требований к ПО,описывающая взаимодействие системы с частью программного обеспечения и содержащая требования к характеристикам и функциональности данной части программного обеспечения.

Спецификация требований к ПОне должна содержать:

· требований к процессу разработки;

· требований к проекту.

Одной из главных причин низкого качества разрабатываемого ПО является недостаточная детализация спецификации и ее низкая формализация. В общем случае ПО создается на основании требований из собственного видения продукта заказчиком. Очевидной причиной возможных ошибок, заложенных в спецификации, является недостаточная работа разработчика с заказчиком.

***Необходимость***- при этом требование определяют существенную способность, характеристику, ограничения или показатель качества.

***Свобода реализаций*** - при этом требование должно быть независимым от его реализации и оно определяет, что требуется, но не как это требование должно быть реализовано.

***Однозначность*** - при этом требование устанавливается таким образом, чтобы оно могло быть интерпретировано только одним способом и сформулировано просто и понятно. Его могут однозначно интерпретировать разработчик и заказчик.

***Непротиворечивость требований*** *-* спецификация требований к ПОнепротиворечива тогда и только тогда, когда каждое заявленное требование не противоречит ни одному другому требованию.

***Полнота*** - при этом считается, что требование определено достаточно полно, если оно не нуждается в дальнейшем пояснении.

***Единичность*** - (атомарность) сформулированное требование должно включать только 1 требование.

***Реализуемость*** - требование реализуемо, если оно с применимым уровнем риска технически достижимо в условиях существующих систем ограничений(например, ограничения по стоимости и срокам разработки).

***Трассируемость*** - требование вверх определяется возможностью определить связь между данным требованием и требованиями вышестоящих в иерарх-и до потребностей конкретных заданного прав-дателей и др источников(анпр. документов, модулей и тд)

Трассируемость требований определяется возможностью отследить связь между требованием и другими артефактами проекта (документами, моделями, текстами программ и т.д. ), например должен быть обеспечен легкий доступ к каждому требованию из документации, создаваемой в процессе разработки продукта.

Требование также должно позволить отследить связь между ним и требованиями в спецификации более низких уровней и другими результатами разработки. Для требований должны быть идентифицированы все родительско-дочерние связи так, чтобы можно было отобразить требования от источника до его реализации.

***Проверяемость (верефицируемость)***- при этом требование проверяемо, если имеется возможность проверки реализации данного требования в с-ме или ПС, при этом также если возможность его проверки данного требования в с-ме, то его проверяемость повышается.

### **Характеристики для оценки наборов требований по ISO/IEC/IEEE 29148:2011. Атрибуты требований.**

***Полнота набора требований -***спецификация требований к ПОявляется полной, когда она удовлетворяет следующим критериям.

1. Содержит все существенные требования, касающиеся функциональных возможностей, выполнения, ограничений проектирования, характеристик или внешних интерфейсов. В частности, любые внешние требования, накладываемые спецификацией системы, должны быть выявлены и рассмотрены.
2. Содержит все реакции программного обеспечения на все возможные типы входных данных во всех возможных ситуациях. Важно определить реакции, как на допустимые значения данных, так и на недопустимые.
3. Содержит все подписи и ссылки на все рисунки, таблицы и диаграммы и определение всех терминов и единиц измерения.

***Непротиворечивость*** - Спецификация требований к ПОнепротиворечива тогда и только тогда, когда каждое заявленное требование не противоречит ни одному другому требованию.

***Возможность реал-ции при заданных средствах*** - при этом полный набор треб-ний, может быть реализован в условиях существования ЖЦ проекта.

***Ограниченность*** - при этом набор требований содержит только требования, необходимые для удовлетворения потребностей пользователей и не содержит избыточности требований

Атрибут предназначен для поддержания анализа требований, при этом повышает понимание треб-ний и описательность треб-ний

1. Идентификация (при этом требование должно быть уникально идентифицируемо(числом, некоторым именем и т.п.)Уникальные идентификаторы помогают в транслировании требований. Однажды назначенный идентификатор никогда не изменяется, даже если само требование изменено. И не может повторно использоваться, даже если требование удалено)
2. Приоритет правообладателей(для каждого требования должен быть определен приоритет, который может быть установлен достижением консенсуса)
3. Зависимость - если сущ зависимость между треб-ями она должна быть опред. При этом связь между треб-ем должна быть такой, чтобы если основн треб-е удалено, то вспомог треб-е тоже могло быть удалено
4. Риск(для каждого треб-я должен быть определены риски связанных с его реал-цией)
5. Источник - кажд треб-е должно вкл атрибут, указ составителя данного треб-я
6. Обоснование - каждое треб-е должно быть обосновано при этом должно объяснять причины необх-ти треб-я и давать советы по его исп-нию в любых исследованиях предметной области моделирования.
7. Трудность - для каждого треб-ния должны быть отличен уровень предполагаемой трудности и его реал -ции
8. Тип - требования различают по смыслу и типу св-в, которые они представляют(для группировки треб-ний для последующего анализа и распределения)

### **Понятие ошибки и отказа ПО. Понятие надежности ПО. Особенности надежности программ по сравнению с надежностью аппаратуры.**

Существуют различные подходы к определению одного из важнейших понятий надёжности ПО – ошибки ПО. Одним из распространённых определений ошибки в ПО является факт расхождения между ПО и его спецификацией. Но такое определение предполагает корректность спецификации, что на практике не всегда верно.

Например, одна из версий ПО Системы противоракетной обороны (ПРО) США восприняла восход луны за вражеский объект. С точки зрения заказчика (Министерства обороны, МО США) это ошибка. Но с точки зрения разработчика – нет, т.к. в соответствии со спецификацией на разработку системы она должна осуществлять перехват любых неопознанных движущихся объектов. Но заказчик (МО США) не мог себе даже представить, что разрабатываемое ПО воспримет луну за вражеский объект.

**Майерс предложил считать, что в ПО есть ошибка, если оно не выполняет того, что пользователю от него разумно ожидать**. Но разумность ожидания пользователя трудно формализовать и тем более измерить.

**Ошибка** в ПО - некоторый дефект (изъян) в разработке ПО, который вызывает несоответствие ожидаемых результатов выполнения ПО и фактически полученных. Дефект может возникнуть на всех стадиях разработки ПО (анализ требований, проектирование, кодирование, ...). Дефектом также может быть любое несоответствие ожиданиям заказчика, которое даже может быть и не определено в спецификации ПО [29].

В общем случае ошибки в ПО неразрывно связаны со своими спецификациями. Очевидно, что чем более формализована спецификация и чем более она детализирована, тем выше качество будущего ПО. Но как показывает анализ разработки программных продуктов, формальные спецификации встречаются редко, потому что:

1. большинство людей предпочитают неформальные спецификации, т.к. их легче понять ( они разрабатываются на естественном языке).
2. написание формальной спецификации дорогостоящее дело.

Дадим следующее определение ошибки, которое будем считать наиболее правильным: будем считать, что в ПО есть ***ошибка***, если его поведение не соответствует своей спецификации при условии, что спецификация корректна (правильна).

***Отказ ПО*** – проявление в ходе эксплуатации скрытой ошибки разработки, приводящей к получению неприемлемого результата или времени выполнения, либо того и другого вместе.

Одни авторы считают, что понятие надежности ПО не аналогично этому понятию в аппаратуре.

Например, Майерс писал: «Надежность ПО – есть вероятность работы без отказов в течении определенного периода времени, рассчитанная с учетом стоимости для пользователя каждого отказа».

Другие считают, что наблюдаемое поведение ПО при его отказах аналогично поведению при отказах аппаратуры. Это объясняется, что исходные данные в процессе эксплуатации ПО выбираются случайным образом, что приводит к случайному моменту проявления скрытых ошибок в ПО.

В соответствии со стандартом ISO/IEC 25010:2011 ***надежность*** – *степень выполнения системой, продуктом или компонентом заданных функций в заданных условиях в течение заданного периода времени* [6].

В отличие от аппаратуры не происходит износа или старения ПО. Снижение надежности ПО происходит из-за ошибок требований, проектирования и реализации. Отказы из-за этих ошибок зависят от способа использования ПО и контекста используемой среды, а не от времени работы.

Характеристике ***надежность*** в соответствии со стандартом ISO/IEC 25010:2011 соответствуют следующие подхарактеристики.

***Завершенность (стабильность***, ***Maturity)*** – степень соответствия системы, продукта или компонента потребностям в надежности при нормальной эксплуатации. Завершенность зависит от количества ошибок, оставшихся в системе, продукте или компоненте, и определяет возможность их безотказной работы.

***Готовность (Availability)*** – степень работоспособности и доступности системы, продукта или компонента тогда, когда требуется их использование.

***Устойчивость к ошибке (Fault tolerance)*** – степень функционирования системы, продукта или компонента в соответствии с предназначением, несмотря на наличие сбоев аппаратного обеспечения или ошибок в программном обеспечении.

***Восстанавливаемость (Recoverability)*** – степень восстановления поврежденных данных и переустановления требуемого состояния системы в случае прерывания или отказа продукта или системы.

Можно выделить следующие ***особенности программного обеспечения*** по сравнению с аппаратным обеспечением компьютеров.

* Ошибки в программах проявляются при их выполнении на некоторых наборах исходных данных (в аппаратуре – из-за сбоев и отказов аппаратуры).
* Компонентам программ не присущи старение и износ.
* Коррекция ошибок в программе приводит к изменению её конфигурации и обычно устраняет саму возможность повторного проявления данной ошибки.
* Тщательный анализ типа отказа и его влияние непрактичны в сложных программах.

### **Процесс разработки ПС в соответствии с СТБ ИСО/МЭК 12207-2003. Причины появления ошибок в ПО.**

### ***Жизненный цикл*** (ЖЦ) – это структура, состоящая из процессов, работ и задач, включающая в себя разработку, эксплуатацию и сопровождение ПС [4].

### Процесс представляет собой набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты.

### В соответствии с данным стандартом, каждый процесс ЖЦ разделен на набор работ, а каждая работа – на набор задач.

### Данный стандарт описывает архитектуру процессов жизненного цикла ПС. При этом он не определяет детали реализации или выполнения работ и задач, входящих в данные процессы. Стандарт не предопределяет конкретную модель ЖЦ или метод разработки ПС.

### Пользователи, использующие данный стандарт, должны сами:

### выбирать модель ЖЦ применительно к разрабатываемому ПС;

### распределять процессы, работы и задачи, выбранные из данного стандарта, на данной модели;

### выбирать и применять методы разработки программных средств и выполнять работы и задачи, соответствующие разрабатываемому ПС.

### Все процессы ЖЦ делятся на следующие ***группы*** [4]:

### основные процессы ЖЦ;

### вспомогательные процессы ЖЦ;

### организационные процессы ЖЦ.

### 

### **Основные процессы** ***ЖЦ***:

### - процесс заказа;

### - процесс поставки;

### - процесс разработки;

### - процесс эксплуатации;

### - процесс сопровождения.

### ***Процесс заказа****.* Содержит работы и задачи, выполненные заказчиком, и состоит из определения потребностей заказчика к системе подготовки и выпуска заявки на подряд, выбор поставщика.

### ***Процесс поставки****.* Состоит из работ и задач, выполняемых поставщиком, начиная с решения о подготовке предложения в ответ заявки на подряд, присланной заказчиком или с подписания договора и вступления с заказчиком в договорные отношения по поставке системы.

### ***Процесс разработки*** содержит 13 работ. Состоит из работ и задач, выполняемых разработчиком и включает в себя работы [4]:

### - по анализу требований;

### - по проектированию;

### - по программированию;

### - по сборке;

### - по тестированию;

### - по вводу в действие;

### - по приёмке программных средств.

### В данный процесс могут быть включены работы, связанные с разработкой системы, если это оговорено в договоре.

### *1-я работа – подготовка процесса*. Должны быть выбраны и адаптированы стандарты, инструментарии и языки программирования для выполнения работ в процессе разработки, так же должна быть выбрана модель ЖЦ.

### *2-я работа – анализ требований к системе*. Система включает в себя объекты технических и программных средств. Должны быть разработаны требования к системе. Требования к системе должны охватывать:

### - функции и возможности системы;

### - коммерческие и организационные требования;

### - требования пользователя;

### - требования безопасности и защиты;

### - эргономические требования;(требования к системе, чтобы чел-оператор не страдал при работе)

### - требования к интерфейсам;

### - эксплуатационные требования;

### - требования к сопровождению.

### Результатом данной работы должна быть документально оформленная спецификация требований к системе (System Requirement Specification).

### *3-я работа – проектирование системной архитектуры*. Должна быть определена общая архитектура всей системы, должны быть указаны объекты технических и программных средств и ручных операций. Ручные операции представляют собой те операции, которые пользователь системы выполняет вручную, так как, в основном, все разрабатываемые системы являются автоматизированными, а не автоматическими. Должно быть обеспечено распределение всех требований между объектами архитектуры системы. Затем на основе указанных объектов архитектуры должны быть определены объекты конфигурации технических и программных средств и ручных операций. Должна быть документально оформлена привязка системной архитектуры и требований к системе относительно установленных требований.

### *4-я работа – анализ требований к программным средствам (ПС)*. Применительно к каждому программному объекту архитектуры должна быть разработана спецификация требований. Должны быть установлены и документально оформлены следующие требования к программным средствам:

### - функциональные требования;

### - технические требования, включая производительность, физические характеристики и окружающие условия, под которые должен быть создан программный объект;

### - требования к внешним интерфейсам программного объекта;

### - квалификационные требования;

### - требования безопасности и защиты;

### - эргономические требования, включая требования, относящиеся к ручным операциям;

### - требования к определению данных и базе данных;

### - требования по вводу в действие и приёмке поставляемого ПС на объектах эксплуатации и сопровождения;

### - требования к документации пользователя;

### - требования к эксплуатации объекта пользователем;

### - требования к обслуживанию пользователя.

### Результатом данной работы должна быть спецификация требований к ПС (SoftwareRequirement Specification).

### *5-я работа – проектирование программной архитектуры*. Требования к программному объекту должны быть преобразованы в программную архитектуру, которая описывает общую структуру программного объекта и определяет компоненты объекта. Затем должно быть обеспечено распределение всех требований к программному объекту между его компонентами. Должны быть разработаны и документально оформлены:

### - общий (эскизный) проект внешних интерфейсов программного объекта и интерфейсов между компонентами объекта;

### - общий (эскизный) проект базы данных;

### - предварительные версии документации пользователя;

### - общие требования к тестированию программного объекта.

### *6-я работа – техническое проектирование ПС*. Должен быть разработан технический проект для каждого компонента программного объекта. Компоненты программного объекта должны быть уточнены на уровне программных модулей. Должно быть обеспечено распределение технических требований к компонентам между программными модулями. Технический проект должен быть документально оформлен. Кроме того, должен быть разработан и документально оформлен технический проект внешнего интерфейса программного объекта, интерфейса между компонентами и интерфейса между программными модулями. Должны быть разработаны и документально оформлены:

### - технический проект базы данных;

### - требования к тестированию и программе испытаний программных модулей.

### *7-я работа – программирование и тестирование ПС*. Должен быть разработан и документально оформлен каждый программный модуль и база данных. Кроме того, должен быть протестирован каждый модуль и база данных на соответствие заданным требованиям. Результаты тестирования должны быть документально оформлены.

### *8-я работа – сборка ПС*. Все модули и компоненты должны быть собраны в единый программный объект и протестированы. Результаты сборки и тестирования должны быть документально оформлены.

### *9-я работа – квалификационные испытания ПС*. Должны быть проведены испытания (тестирование) на соответствие квалификационных требований программному объекту. При тестировании должна быть проверена правильность выполнения каждого требования. Ошибки фиксируются и документально оформляются.

### *10-я работа – сборка системы*. Программные объекты конфигурации должны быть собраны в единую систему вместе с объектами технической конфигурации и при необходимости с другими системами.

### *11-я работа – квалификационное испытание системы*. Должны быть проведены в соответствии с требованиями, установленными к системе (эталон – спецификация требований к системе). Результаты квалификационных испытаний документально оформляются.

### *12-я работа – ввод ПС в действие*.

### *13-я работа – обеспечение приемки ПС*.

### ***Процесс эксплуатации***. Данный процесс состоит из следующих работ:

### - подготовка процесса;

### - эксплуатационные испытания;

### - эксплуатация системы;

### - поддержка пользователя.

### ***Процесс сопровождения***. Выполняется персоналом сопровождения и реализуется при модификациях ПС. Целью данного процесса является изменение существующего ПС при сохранении целостности. Так же процесс охватывает вопрос снятия ПС с эксплуатации.

### 

### **Вспомогательные процессы ЖЦ ПС** [4]**:**

### - документирование;

### - управление конфигурацией;(поддержанием целостности системы на протяжении всего ЖЦ)

### - обеспечение качества;

### - верификация;(правильно ли создана с-ма)(проверяет соответствие ПО системной спецификации, в частности функциональным и нефункциональным требованиям)

### - аттестация;(правильно ли работает с-ма)(необходимо убедиться, что программный продукт соответствует ожиданиям заказчика)

### - совместный анализ;

### - аудит;

### - решение проблем;

### ***Документирование*** – процесс формализованного описания информации, созданной в процессе или работе ЖЦ ПС.

### ***Управление конфигурацией*** – процесс применения административных и технических процедур на всем протяжении ЖЦ программных средств.

### ***Обеспечение качества*** – процесс обеспечения соответствующих гарантий того, что программные средства и процессы в ЖЦ проекта соответствуют установленным требованиям и утвержденным планам.

### ***Верификация*** – процесс определения того, что программное средство функционирует в полном соответствии с требованиями или условиями, реализованными в предшествующих работах. Верификация может применяться не только к ПС, но и к любым другим результатам работы. Например, верификации могут подвергаться документация, программная архитектура и т.д.

### ***Аттестация*** – процесс аттестации является процессом определения полноты соответствия установленных требований созданной системы или ПС их функциональному назначению. Аттестации могут подвергаться любые промежуточные продукты процесса обработки, но чаще аттестации подвергаются продукты на завершающей стадии обработки.

### ***Совместный анализ*** – процесс оценки состояний и результатов работ по всему проекту. Совместные анализы проводятся в течении всего ЖЦ договора и применяются как на уровне управления проектом, так и на уровне его технической реализации.

### ***Аудит*** – процесс определения соответствия требованиям, планам и условиям договора.

### ***Решение проблем*** – процесс анализа и решения проблем, которые обнаружены в ходе выполнения разработки, эксплуатации, сопровождения или других процессов.

### 

### **Организационные процессы ЖЦ** [4]**:**

### - управления;

### - создания инфраструктуры;

### - усовершенствования;

### - обучения.

### ***Процесс управления*** состоит из работ и задач, которые могут быть использованы любой стороной, управляющей соответствующим процессом (планирование, выполнение, контроль, оценки).

### ***Процесс создания инфраструктуры*** является процессом установления и обеспечения (сопровождения) инфраструктуры, необходимой для любого другого процесса. Инфраструктура может содержать технические и программные средства, методики, стандарты и условия для разработки, эксплуатации и сопровождения.

### ***Процесс усовершенствования*** является процессом установления оценки измерения, контроля и улучшения любого процесса жизненного цикла ПС.

### ***Обучение*** – процесс обеспечения первоначального и продолженного обучения персонала работам по заказу, поставке, разработке, эксплуатации или сопровождению ПС.

### Одним из распространенных определений ошибки является расхождение между спецификацией требований и ПО. Такое определение предполагает корректность спецификации требований, что не всегда так.

### Ошибка в ПО представляет некоторый дефект в его разработке, который вызвал несоответствие ожидаемым результатам выполнения ПО. Дефект может возникнуть практически во всех работах процесса разработки.

### В классической теории надёжности систем*,* надёжность любого объекта закладывается при проектировании, реализуется при изготовлении и расходуется при эксплуатации. При этом жизненный цикл объекта состоит из трёх этапов: проектирование, изготовление и эксплуатация. В общем, ПО относится к сложным объектам. Оно имеет по сравнению с разработкой аппаратуры как схожие моменты, так и свои особенности.

### Например, в 4-й работе процесса разработки ЖЦ ПС [4] дефект – отсутствие требований или их неверная трактовка.

### В 5-й работе процесса разработки ЖЦ ПС правильные требования могут быть преобразованы в неправильную программную архитектуру.

### В 6-й работе процесса разработки ЖЦ ПС правильная программная архитектура может быть преобразована в неправильный технический проект, содержащий алгоритмы и модульную структуру ПС.

### В 7-й работе процесса разработки ЖЦ ПС может быть внесен огромный спектр программных ошибок (зацикливание, отсутствие предварительной инициализации переменных и т.д.).

### В ходе собственно тестирования в 7, 8, 9, 11-й работах процесса разработки ЖЦ ПС могут внестись новые ошибки. Статистика говорит, что исправление каждой 3-й ошибки вносит новую. Главные причины внесения ошибок при тестировании – плохое документирование проекта, низкий уровень характеристики сопровождаемости ПС.

### Как показывает анализ процесса разработки ПО, важнейшей причиной появления ошибок (дефектов, багов) в ПО является неправильный перевод информации из одного представления в другое. Например, требования к ПС преобразуется архитектуру ПС, архитектура ПС преобразуется в техническое проектирование (алгоритм), и т.п.

### В общем, разработку ПО можно просто описать как ряд процессов перевода, начинающаяся с перевода исходной задачи в различные промежуточные решения, заканчивающихся большим набором команд (операторов), выполняющихся на компьютере и решающих данную задачу. Когда не удается полно и точно перевести представление исходной задачи или промежуточного решения в другое более детальное решение, тогда и возникают ошибки в ПО.

### **Методы обеспечения надежности ПО.**

Понимание того, что именно ошибки перевода являются основной причиной ошибок в ПО, является ключом к пониманию проблемы надёжности ПО. Такое понимание явилось основой к разработке подходов к обеспечению надёжности ПО. Данные подходы в основном направлены на предотвращение появления ошибок в процессе разработки ПО и на обнаружение ошибок перевода, возникающих на каждом этапе.

Методы обеспечения надёжности можно разбить на три группы:

1) Предупреждение ошибок;

2) Обнаружение и исправление ошибок;

3) Обеспечение устойчивости к ошибкам;

1)**Предупреждение ошибок**

К первойгруппе относятся методы, позволяющие максимально уменьшить количество ошибок в программах [29]. Очевидно, что единственная важная причина ошибок в ПО – некорректный перевод информации из одного представления в другое. Большинство методов первой группы фокусируются на отдельных процессах перевода и направлены на предупреждение ошибок в этих процессах.

Сложность – основная причина ошибок перевода и таким образом одна из главных причин ненадёжности ПО. Она почти не поддается ни точному определению, ни измерению. Но можно сказать, что мерой сложности объекта является количество интеллектуальных усилий, необходимых для понимания этого объекта.

В общем сложность является функцией взаимодействия между его компонентами. Например, сложность структуры ПС определяется связями между компонентами ПС. Сложность отдельного модуля определяется связями между операторами (командами модуля) и т.п.

Для снижения структурной сложности ПС широко используются два принципа:

1) Независимость компонентов (модулей) ПС;

2) Иерархическая структура (иерархия) ПС.

В соответствии с концепцией независимости с целью минимизации сложности необходимо максимально усилить независимость модулей ПО.

В общем, чтобы уменьшить сложность ПО, нужно разбить его (декомпозировать) на множество небольших достаточно простых для понимания независимых модулей. Высокой степени независимости модулей можно достичь с помощью двух методов понижения структурной сложности:

1) Усилением внутренних связей в каждом модуле

2) Ослаблением взаимосвязи между модулями.

Иерархия позволяет декомпозировать ПО на компоненты с учетом ограничений на их сложность. Каждый вышестоящий уровень представляет собой совокупность структурных отношений между элементами нижестоящих уровней.

Концепция уровня позволяет понять ПО, скрывая несущественные уровни детализации. Например, иерархия в IDEFO. Функциональная модель имеет иерархическую структуру, состоящую из множества IDEFO-диаграмм. Каждая диаграмма любого уровня может быть декомпозирована на несколько диаграмм нижележащего уровня. При этом, каждая из диаграмм должна содержать не более шести блоков из-за ограничений по сложности. Таким образом сложность диаграмм должна находится в пределах понимания данной диаграммы. Иерархия и ограничение сложности на каждом уровне модели позволяют заказчику понять модель, а разработчику сделать меньше ошибок при разработке функциональной модели предметной области.

В первом методе понижения структурной сложности усиление внутренних связей в каждом модуле достигается за счет увеличения связности модуля. Связность модуля определяется как мера зависимости его частей. При этом, чем выше связность, тем больше отдельные части модуля зависят друг от друга.

Важнейшим методом увеличения независимости модулей является ослабление связи между ними. Мерой ослабления связи между модулями является сцепление модулей, которое является мерой относительной независимости модулей. Сцепление модулей является мерой взаимодействия модулей по данным и характеризуется как способом передачи данных, так и свойствами этих данных. Сцепление влияет на сохранность данных при модификациях. Слабое сцепление определяет высокий уровень независимости модулей. Независимые модули могут быть модифицированы без переделки других модулей. В общем два модуля являются полностью независимыми, если в каждом из них не используется никакая информация о другом модуле.

Таким образом максимально независимый модуль имеет максимальную связность и минимальное сцепление.

Если рассматривать программу как набор предложений, связанных между собой некоторыми отношениями (как по выполняемым функциям, так и по обрабатываемым данным), то главная задача, стоящая перед разработчиками, это распределить эти предложения между модулями, таким образом, чтобы:

1) Предложения в каждом модуле были максимально связаны;

2) Связь между любой парой предложений в разных модулях была минимальна.

При проектировании нужно стремиться, во-первых, реализовать отдельные функции отдельными модулями (высокая связность), во-вторых, ослаблять связи между модулями по данным, применяя передачу параметров (слабое сцепление модулей).

Таким образом, очевидно, что предупреждения возникновения ошибок представляет собой оптимальный путь к достижению НПО. Но гарантировать отсутствие ошибок в общем невозможно.

2)**Обнаружение и исправление ошибок**

Данная группа методов обеспечения надежности основывается на предположении о том, что, несмотря ни на что, ошибки в ПО всё же будут.

Тестирование и отладка являются основным средством обнаружения и устранения ошибок для обеспечения надежности ПО.

Наибольшие усилия затрачиваются во время тестирования и отладки ПО. Под отладкой понимается процесс внесения изменения в ПО с целью обеспечения его соответствия заданным требованиям. В начале отладки производится проверка ПО с помощью различных методов тестирования. Так как исчерпывающее тестирование практически невозможно, то нет полной гарантии того, что после тестирования ПО не осталось ошибок. Можно так же отметить, что само по себе тестирование не является методом повышения надёжности, т.к. только позволяет лишь выявить имеющиеся в ПО ошибки. Если же по результатам тестирования будут приняты меры к устранению найденных ошибок, то только тогда будет повышена надёжность ПО.

В общем, использование методов первой и второй групп позволяет существенно уменьшить количество ошибок в ПО, но не может их полностью ликвидировать. Поэтому важной и актуальной является задача создания ПО, устойчивого ко всем *факторам ненадёжности*:

1) аппаратным, обусловленным влиянием сбоев и отказов аппаратуры;

2) программным, обусловленным влиянием ошибок в программах;

3) внешним, вследствие искажений в исходных данных, поступающих на вход программ.

3) **Устойчивость к ошибкам в ПО**

Данная группа методов обеспечения надежности также основывается на предположении о том, что, несмотря ни на что, ошибки в ПО всё же будут.

Методы этой группы ставят своей задачей обеспечить функционирование ПО при наличии в нем ошибок. Устойчивость ПО достигается за счёт внесения в него различных форм избыточности:

- Временной;

- Информационной;

- Программной.

Например, программная избыточность может обеспечиваться использованием метода N-версионного программирования, когда независимо создаётся N версий разрабатываемого ПО разными коллективами. Но объективный недостаток данного метода – увеличение стоимости разработки ~ в N раз.

### **Модель Джелинского-Моранды. Определение характеристик модели с помощью метода максимального правдоподобия.**

Данная модель была предложена Джелинским и Морандой [29, 36].

*Допущения модели*:

1. время до очередного отказа распределено по экспоненциальному закону;

2. все ошибки равновероятны и их появление не зависит друг от друга;

3. частота появления ошибок (интенсивность отказов, в англоязычной литературе так же имеет название *функция риска*) пропорциональна числу не выявленных ошибок:

***l(ti)=Kjm·[E0-(i-1)]*** ,

где: ***E0*** – число ошибок в ПО до начала тестирования и отладки; ***Kjm*** – коэффициент Джелинского-Моранды; ***ti*** – интервал времени между (***i-1)***-й и ***i***-й обнаруженными ошибками; ***i*** *–* число ошибок, обнаруженных к моменту отладки ***ti***;

4. ***l(ti) = const***на интервале между двумя смежными моментами появления ошибок;

5. каждая обнаруженная ошибка в ПО немедленно устраняется и число оставшихся ошибок уменьшается на 1;

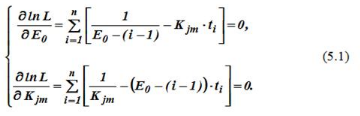
6. ошибки корректируются без внесения новых ошибок.

Время ***ti*** соответствует длительности выполнения ПО на компьютере и не учитывает простои компьютера для анализа результатов и их корректировки.

На рис. 5.1 представлена зависимость интенсивности отказов от времени.

|  |  |
| --- | --- |
|  | Рис. 5.1. Зависимость интенсивности отказов от времени  На данном рисунке:    ***λ(t***i***) = K***Jm ***[E***0 ***– (i-1)]***  ***λ(t***1***) = K***Jm ***[E***0 ***– (1-1)] = K***Jm***E***0  ***λ(t***2***) = K***Jm ***[E***0 ***– (2-1)] = K***Jm ***[E***0 ***– 1]***  ***λ(t***3***) = K***Jm ***[E***0 ***– (3-1)] = K***Jm ***[E***0 ***– 2]***  ***λ(t***4***) = K***Jm ***[E***0 ***– (4-1)] = K***Jm ***[E***0 ***– 3]***    Так как согласно допущению 1 время до очередного отказа распределено по экспоненциальному закону, то вероятность безотказной работы, т.е. отсутствия следующего ***i***-го отказа равна: |
|  | то плотность вероятности отказов равна:    *Среднее время безошибочной работы или средняя наработка до следующего* ***i****-ого отказа* определяется выражением:    Параметрами модели (исходными данными) является статистика об ошибках, т.е. интервалы времени между отказами.  *Характеристиками модели* (то, что нужно определить) являются:   * ***P(ti)*** – вероятность отсутствия следующего ***i***-ого отказа; * – среднее время до очередного i-ого отказа; * ***Kjm*** – коэффициент Джелинского-Моранды; * ***E0*** –число ошибок в ПО до начала тестирования и отладки.   Характеристики модели определяются при помощи ***метода максимального правдоподобия***.  **Пример.**  Оценка характеристик модели Джелинского-Моранды ***E0*** и ***Kjm*** осуществляется при помощи метода максимального правдоподобия.  При этом фиксировались интервалы времени между отказами вследствие ошибок в ПО {ti | i = 1, n} случайной величины ***Т*** – времени между отказами ПО.  Пусть интервалы времени между отказами ПО равны  ***t1=10 час.; t2=15 час.; t3=20 час.; t4=25 час.***  Всего обнаружено ***n = 4*** ошибки.  Введём понятие о функции правдоподобия ***L*** как о совместной ***n***-мерной плотности распределения вероятностей события, заключающегося в том, что в процессе тестирования ПО случайная величина ***T***примет значение ***t1=10 час.; t2=15 час.; t3=20 час.; t4=25 час.***  Функция правдоподобия согласно допущению 2 будет иметь вид:  L(t1, t2, t3, t4) = П{i=1, n}f(ti)  или, подставив значения функции плотности,    В качестве оценок неизвестных параметров ***Kjm*** и ***E0*** выбираются такие функции  , которые максимизируют функцию правдоподобия. Затем, на основании известных правил дифференциального исчисления для нахождения оценок максимального правдоподобия ***Kjm*** и ***E0*** составляется система уравнений    и выбирается то решение, которое обращает функцию правдоподобия в максимум. Так как экстремум функций ***L*** и ***ln L*** достигается при одних и тех же значениях ^***Kjm*** и ^***E0*** , то для упрощения вычислений пользуются логарифмической функцией правдоподобия.  Возьмём натуральный логарифм от функции правдоподобия, чтобы произведение функций заменить суммой: |

Возьмём частные производные по ***E0*** и ***Kjm*** и найдём условия экстремума:



Полагая ***Kjm*** = ^***Kjm*** , ^***E0*** = ***E0***  и решая с-му ур-ний(5.1) получаем:

а) оценка коэффициента Джелинского-Моранды

^***Kjm*** = 0.017

б) оценка числа ошибок в ПО до начала тестирования

^***E0*** =5

Интенсивность отказов до следующей пятой ошибки



Средняя наработка до обнаружения следующей пятой ошибки



Вероятность безотказной работы, т.е. отсутствия следующего пятого отказа



**Критический анализ модели Джелинского-Моранды.**

Многие допущения модели спорны. Например:

- что все ошибки одинаково серьёзны;

- ошибка исправляется немедленно (или ПО не используется до тех пор, пока найденная ошибка не будет исправлена);

- ошибки корректируются без внесения новых ошибок;

- основное допущение - то, что для всех программ ***λ(t***i***)*** имеет вид, представленный на рис. 5.1, и ***λ(ti)***= const между появлениями ошибок, также спорно.

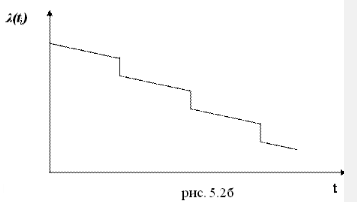
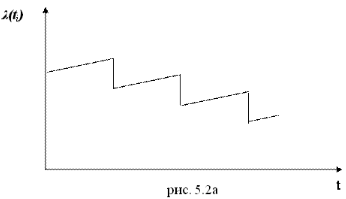
При этом каждая ошибка уменьшает ***λ(ti)*** на постоянную величину ***KJM***.

Желание выразить надёжность ПО некоторой функции времени в общем разумно. Но надо понимать, что в действительности (напрямую) надежность ПО от времени не зависит. Аппаратура напрямую зависит от времени, например, аппаратура стареет. НПО является функцией числа ошибок, их серьёзности и их расположения (нахождения), а также того как ПО используется. Сторонники постоянства ***λ(ti)*** соглашаются с тем, что входные данные ПО не являются случайными и равновероятными, но утверждают, что входные данные выглядят как случайные (т.е. близки к случайным) так как область их изменения велика.

Однако, другие утверждают, что между ошибками ***λ(ti)*** возрастает (рис. 5.2а). Они аргументируют это тем, что входные данные ПО постепенно становятся ближе к тем данным, которые обнаружат оставшиеся ошибки.

Есть и такие, кто утверждает, что ***λ(ti)*** уменьшается со временем (рис. 5.2б), мотивируя это тем, что чем больше ПО выполняется без шибок, тем меньше вероятность того, что одна из них будет обнаружена.

Есть также и такие, кто утверждает, что ***λ(ti)*** между ошибками уменьшается и возрастает всякий раз, когда ошибка обнаруживается (рис. 5.2в). Они этот факт аргументируют тем, что при обнаружении ошибки возрастает вероятность появления ещё одной ошибки. Этот факт они постулируют как аксиому.



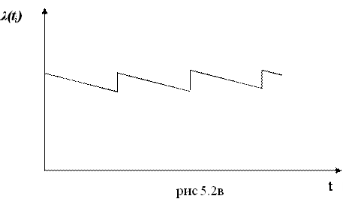


Рис. 5.2. Другие функции интенсивности отказов

Все эти гипотезы о поведении ***λ(ti)*** на рисунках 5.1; 5.2а; 5.2б; 5.2в демонстрируют в общем некоторый рост надёжности ПО. Но как показывает практика в реальности ПО это не всегда имеет место, даже в случаях, когда исправления не увеличивают число ошибок. Интенсивность отказов (частота) в больших проектах при измерении в течении нескольких лет в общем может иметь тенденцию к понижению ***λ(ti)***, т.е. росту надёжности ПО, но внутри этого периода возможны колебания. Анализ частоты обнаружения ошибок при тестировании 14 проектов показал, что частота обнаружения ошибок в 5 проектах достигла пика в начале работы, в 5 – в середине, и в 4-х в конце работы [29].

В общем можно утверждать, что каждая программа имеет своё собственное уникальное распределение ***λ(ti)***. Даже, при каждой установке каждой программы ***λ(ti)*** – своё. Можно также отметить, что функция ***λ(ti)*** для одной и той же программы может меняться со временем или при обнаружении каждой ошибки.

Как показывает практика, модель JM устойчиво даёт завышенное число остаточных ошибок.

### **Модель Шика-Волвертона. Определение характеристик модели с помощью метода максимального правдоподобия.**

Данная модель является модификацией модели Джелинского–Моранды и была предложена Шиком и Волвертоном [23, 29, 36].

Дополнительно к допущениям модели Джелинского–Моранды используется следующее *допущение*: частота появления ошибок пропорциональна времени отладки программы ***ti***, т.е. вероятность обнаружения ошибок с течением времени должна возрастать.



где:

***E0*** – число ошибок в ПО до начала тестирования и отладки;

***Ksw*** – коэффициент Шика-Волвертона;

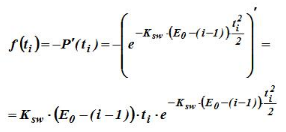
***ti*** – интервал времени между (***i-1)***-й и ***i***-й обнаруженными ошибками;

***i*** *–* число ошибок обнаруженных к моменту отладки ***ti***.

Для вероятности безотказной работы имеем следующее выражение:



Функция плотности будет иметь вид:



**Пример.**

Оценка характеристик модели Шика-Волвертона ***E0*** и ***Ksw*** осуществляетсяпри помощи метода максимального правдоподобия.

Пусть интервалы времени между отказами

***t1=7 час.; t2=12 час.; t3=23 час.; t4=17 час.; t5=10 час..***

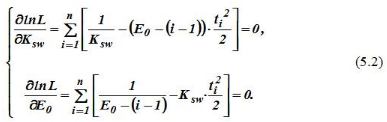
Пусть число найденных ошибок ***n = 5.***

Функция правдоподобия будет иметь вид:

Возьмём натуральный логарифм от функции правдоподобия, чтобы произведение функций заменить суммой:



Возьмём частные производные по ***E0*** и ***Ksw*** и найдём условия экстремума:



Решая с-му ур-ний (5.2) и полагая n=5, ***Ksw*** = ^***Ksw***, ^***E0*** = ***E0*** ,

^***Ksw***= 0.001

и оценку числа ошибок

^***E0*** = ***11***

***Рассмотренные модели Джелинского-Моранды, Шика-Волвертона можно использовать как на этапе тестирования и отладки, так и на этапе эксплуатации.***

### **Геометрическая модель надежности ПО. Определение характеристик модели с помощью метода максимального правдоподобия.**

Рассмотрим геометрическую модель надёжности ПО, предложенную Морандой.

В данной модели предполагается, что исходное число ошибок в программе – величина не фиксированная, более того все ошибки не равновероятны. Считается также, что по мере отладки обнаруживать ошибки становится все труднее, таким образом, ПО никогда не освобождается от ошибок. Основные *исходные предпосылки* для этой модели следующие:

1. общее число ошибок неограниченно;

2. обнаружение ошибок не равновероятно;

3. обнаружение ошибок – процесс, независимый от ошибок;

4. ПО работает в условиях, близких к реальным;

5. интенсивность обнаружения ошибок образует геометрическую прогрессию и она в интервале между появлениями ошибок постоянна.

Интенсивность отказов имеет следующий вид:



где: ***λ*** ***(0) = D*** – исходное значение интенсивности отказов;

***K*** – константа пропорциональности ***0 < K < 1;***

***ti*** – время между появлениями (***i – 1***)-ой и ***i***-ой обнаруженных ошибок.

**Пример.**

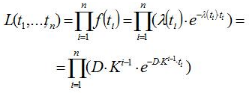
Нахождение характеристик геометрической модели методом максимального правдоподобия.

Пусть интервалы времени между появлениями ошибок

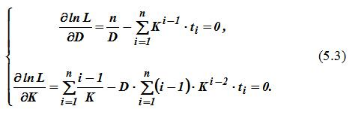
***t1=7 час.; t2=12 час.; t3=23 час.; t4=17 час.; t5=10 час.***

Число ошибок ***n = 5.***

Функция правдоподобия примет вид:



Прологарифмировав и взяв частные производные по ***D*** и ***K***, получим условия экстремума:



Полагая n=5, D=^D, K=^K и решая систему уравнений (5.3), получаем оценки максимального правдоподобия

^K=0.892

^D=0.092

Среднее время до обнаружения ***(n+1)***–ой ошибки:



Значения интенсивности отказов на каждом интервале тестирования приведены в таблице 5.1.



### **Модель Миллса.**

Предложил Х.Д.Миллс. В модели не делается никаких предположений о поведении интенсивности отказов **λ(t).** Модель строится на статистическом фундаменте. Сначала программа “засоряется”некоторым известным количеством ошибок.

Пусть в программном обеспечении внесено S ошибок случайным образом. Затем делается предположение, что для собственных и внесенных ошибок вероятность их обнаружения при последующем тестировании одинакова. Тестируя программное обеспечение в течении некоторого времени и отсортировывая собственные и внесенные ошибки можно оценить первоначальное собственное число ошибок в программном обеспечении N.

Пусть при тестировании обнаружено n + V ошибок, где n – число найденных собственных ошибок; V– число найденных внесенных ошибок.

Так как внесенные ошибки вносятся в программное обеспечение случайным образом и на основании предположения о том, что вероятность обнаружения собственных и внесенных ошибок одинакова и зависит только от их количества, то можно считать, что отношение будет равно отношению Тогда первоначальное число ошибок в программном обеспечении оценивается соотношением:



Например, если в программу внесено S = 20 ошибок и к некоторому моменту тестирования обнаружено n = 15 собственных и V = 5 внесенных ошибок, то оценка N = 20 \* 15/5 = 60 ошибок.

Вторая часть модели связана с выдвижением и проверкой гипотез об первоначальном числе ошибок в программе, т.е. собственном числе ошибок N. Пусть в программе имеется не более K собственных ошибок и дополнительно внесем в нее S ошибок. Затем программа тестируется до тех пор, пока не будут обнаружены все внесенные ошибки S. При этом подсчитывается число обнаруженных собственных ошибок n.

Затем вычисляется мера доверия к модели, т.е. вероятность того, что модель будет правильно отклонять ложное предположение, т.е. вероятность того, что в программе содержится K ошибок:



Например, если мы утверждаем, что в программе нет ошибок, то K = 0. Затем вносим в программу, например, S = 4 ошибки и тестируем ее. Во время тестирования обнаруживаем все внесенные S = 4 ошибки. При этом не обнаруживаем ни одной собственной ошибки, т.е. n = 0. Тогда, по условию n≤K (т.е. 0 ≤ 0)



Таким образом, с вероятностью 0.8 можно утверждать, что в программе нет ошибок (K = 0).

Если мы хотим повысить меру доверия к данной гипотезе (K=0), то необходимо повысить число вносимых ошибок S в программу. Пусть S=19 ошибок. Тестируем программу до тех пор, пока не будут обнаружены все S=19 внесенных ошибок. При этом не обнаруживается ни одной собственной ошибки, т.е. n=0.

Тогда



Т.е. с вероятностью 0.95 можно утверждать, что в программе нет ошибок, т.е. K=0.

Пусть мы утверждаем, что в программе не более трех собственных ошибок, т.е. K=3, и вносится S=6 ошибок. Тестируем программу до тех пор, пока не будут обнаружены все S=6 ошибок. При этом обнаруживается не более трех собственных ошибок, т.е. n≤3.

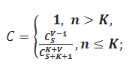
Тогда



и с вероятностью 0.6 можно утверждать, что в программе не более K=3 собственных ошибок.

Но если в процессе тестирования будет обнаружено n>K, то гипотеза о том, что собственное найденное число ошибок n будет меньше или равно K, отвергается.

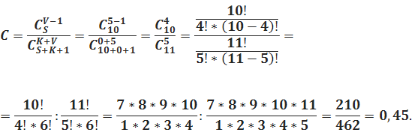
Таким образом, выражения (5.4) и (5.5) образуют модель ошибок *N*. Выражение (5.4) предсказывает число ошибок *N*, а выражение (5.5) определяет меру доверия *C* к модели ошибок. Недостатком модели ошибок является то, что вероятность гипотезы не будет определена до тех пор, пока не будут обнаружены во время тестирования программы все внесенные ошибки. В общем все внесенные ошибки в программу могут до конца этапа тестирования и не быть обнаруженными. Чтобы справиться с данной трудностью, необходимо модифицировать выражение (5.5) для *C* таким образом, чтобы *C* можно было оценивать после того, как найдено *V* внесенных ошибок. При этом *V ≤ S*.



число сочетаний из S по V-1;

 число сочетаний из S+K+1 по K+V.

Пусть в программе нет ошибок (K=0), а к моменту оценки надежности обнаружено V=5 из S=10 внесенных ошибок. При этом не обнаружено ни одной собственной ошибки, т.е. n=0. Тогда вероятность того, что в программе нет ошибок, определяется по выражению (5.6) и будет равна:



С вероятностью C=0,45 можно утверждать, что в программе нет ошибок (K=0).

Пусть в программе нет ошибок (K=0), а к моменту оценки надежности обнаружено V=8 из S=10 внесенных ошибок. При этом не обнаружено ни одной собственной ошибки, т.е. n=0. Тогда вероятность того, что в программе нет ошибок, определяется по выражению (5.6) и будет равна: C=0,7.

С вероятностью C=0,7 можно утверждать, что в программе нет ошибок.

Пусть K=0 и обнаружено V=10 из S=10 внесенных ошибок. При этом не обнаружено ни одной собственной ошибки, т.е. n=0. Тогда по выражению (5.6):



С вероятностью C=0,91 можно утверждать, что в программе нет ошибок. В последнем случае, когда V=10, *C* можно оценить с помощью выражения (5.5):



***Достоинства модели:***

1. Модель Миллса одновременно математически проста и интуитивно понятна.

2. Данная модель может оказывать положительное психологическое влияние на группу тестирования, т.к. при обнаружении не всех внесенных ошибок тестировщик уверен в том, что программа содержит еще ошибки и продолжает дальнейшее тестирование.

***Недостаток модели:*** Процесс внесения ошибок является самым слабым местом модели, т.к. предполагается, что собственные и внесенные ошибки обнаруживаются с одинаковой вероятностью. Из этого следует, что внесенные ошибки должны быть типичными для данной программы. Но сложность состоит в том, что неизвестно, какой должна быть типичная ошибка, если тестирование проводит не тот, кто писал программу.

### **Модель надежности ПС в соответствии с ГОСТ 28195-99.**

*ГОСТ 28195–99* определяет ***оценку качества*** программного средства как совокупность операций, включающих выбор номенклатуры показателей качества оцениваемого программного средства, определение значений этих показателей и сравнение их с базовыми значениями. В соответствии с данным стандартом оценка качества должна проводиться применительно ко всем работам жизненного цикла ПС при планировании показателей качества ПС, контроле качества в процессе разработки, проверке эффективности модификации ПС в процессе сопровождения.

Стандартом *ГОСТ 28195–99* рекомендован метод интегральной оценки качества ПС, основанный на иерархической модели качества.

Надежность является одной из шести основных характеристик качества ПС. С учетом этого рассмотрим данный метод оценки в применении к надежности ПС.

В соответствии с данным методом выбор номенклатуры показателей надежности для конкретного программного средства осуществляется с учетом его назначения и требований области применения в зависимости от принадлежности ПС к тому или иному подклассу, определяемому *общесоюзным классификатором продукции (ОКП)*. В стандарте *ГОСТ 28195–99* приведена классификация ПС на подклассы в соответствии с ОКП.

Оценка надежности ПС производится с учетом фазы жизненного цикла (ЖЦ). Фазы жизненного цикла представляют собой совокупности процессов или работ стандарта *СТБ ИСО/МЭК 12207-2003*.

Оценка надежности ПС заключается в выборе номенклатуры показателей, их оценке и сопоставлении с базовыми значениями.

Под ***показателем качества продукции*** в *ГОСТ 28195–99* подразумевается количественная характеристика одного или нескольких свойств продукции, составляющих ее качество, рассматриваемая применительно к определенным условиям ее создания и эксплуатации или потребления.

Обеспечение и оценка качества ПС должны выполняться в течение жизненного цикла программных средств.

Основу описываемого метода оценки надежности составляет *четырехуровневая иерархическая модель надежности. ГОСТ 28195–99* предлагает следующую ***терминологию*** для показателей качества (в том числе и надежности) каждого уровня:

*уровень 1 -*  факторы качества (в терминологии, принятой в международных стандартах, соответствуют характеристикам качества, в данном случае это относится к надежности));

*уровень 2 -*  критерии качества (в международной терминологии – подхарактеристики качества, в данном случае надежности);

*уровень 3 -*  метрики (соответствует международной терминологии);

*уровень 4 -*  оценочные элементы или единичные показатели (данный уровень в международных стандартах отсутствует).

Для фактора надежности составляется четырехуровневая иерархическая модель, отражающая ее взаимосвязь с критериями, метриками и оценочными элементами. Вид данной модели зависит от фазы жизненного цикла ПС.

В качестве примера на рис. 6.1 – 6.3 приведены три верхних уровня иерархической модели фактора надежности для различных фаз жизненного цикла. Номера на данных рисунках соответствуют номерам метрик данного фактора.



Рис. 6.2. Модель надежности для фазы проектирования



Рис. 6.2. Модель надежности для фаз реализации, тестирования,

изготовления и сопровождения

Выбор оценочных элементов в метрике зависит от функционального назначения ПС и формируется с учётом данных, ранее полученных при проведении испытаний ПС и эксплуатации аналогичных программ. Для выбора оценочных элементов ГОСТ 28195-99 предлагает перечень таблиц, содержащих наименование элемента, метод оценки и применяемость элемента для различных подклассов ПС.

Таблица 6.1 содержит перечень оценочных элементов для фактора надёжности. В данной таблице код оценочного элемента состоит из пяти символов. Первый символ (буква) указывает на принадлежность элемента фактору (Н - Надёжность). Два следующих символа – номер метрики, которой принадлежит оценочный элемент (для надёжности номера метрик обозначены на рис. 6.1 – 6.3). Четвёртый и пятый символы – порядковый номер данного оценочного элемента в метрике.

Таблица 6.1 Оценочные элементы фактора надёжности

Входные данные

|  |  |  |  |
| --- | --- | --- | --- |
| Код элемента | Наименование | Метод оценки | Оценка |
| Н0101 | Наличие требований к программе по устойчивости функционирования при наличии ошибок во входных данных | Экспертный | 0 – 1 |
| Н0102 | Возможность обработки ошибочных ситуаций | – " – | 0 – 1 |
| Н0103 | Полнота обработки ошибочных ситуаций | – " – | 0 – 1 |
| Н0104 | Наличие тестов для проверки допустимых значений входных данных | – " – | 0 – 1 |
| Н0105 | Наличие системы контроля полноты входных данных | – " – | 0 – 1 |
| Н0106 | Наличие средств контроля корректности входных данных | – " – | 0 – 1 |
| Н0107 | Наличие средств контроля непротиворечивости входных данных | – " – | 0 – 1 |
| Н0108 | Наличие проверки параметров и адресов по диапазону их значений | – " – | 0 – 1 |
| Н0109 | Наличие обработки граничных результатов | – " – | 0 – 1 |
| Н0110 | Наличие обработки неопределенностей |  | 0 – 1 |
| Н0201 | Наличие требований к программе по восстановлению процесса выполнения в случае сбоя операционной системы, процессора, внешних устройств | СБОИ | 0 – 1 |
| Н0202 | Наличие требований к программе по восстановлению результатов при отказах процессора, операционной системы | – " – | 0 – 1 |
| Н0203 | Наличие средств восстановления процесса в случае сбоев оборудования | – " – | 0 – 1 |
| Н0204 | Наличие возможности разделения по времени выполнения отдельных функций программ | – " – | 0 – 1 |
| Н0205 | Наличие возможности повторного старта с точки останова |  | 0 – 1 |
| Н0301 | Наличие централизованного управления процессами, конкурирующими из-за ресурсов | РЕАЛ-ЦИЯ УПР-Я СР-МИ ВОССТ-Я | 0 – 1 |
| Н0302 | Наличие возможности автоматически обходить ошибочные ситуации в процессе вычисления | – " – | 0 – 1 |
| Н0303 | Наличие средств, обеспечивающих завершение процесса решения в случаях помех | – " – | 0 – 1 |
| Н0304 | Наличие средств, обеспечивающих выполнение программы в сокращенном объеме в случае ошибок или помех |  | 0 – 1 |
| Н0305 | Показатель устойчивости к искажающим воздействиям | Регистрационный  +  Расчетный | где: ***D*** – число экспериментов, в которых искажающие воздействия приводят к отказу;  ***К*** – число экспериментов, в которых имитируются искажающие воздействия;  ***Y*** – вид искажающего воздействия |
| Н0401 | Вероятность безотказной работы ***Р*** | ФУНКЦ-Е В ЗАДАННЫХ РЕЖИМАХ  Регистрационный  +  Расчетный | где: ***N*** – число экспериментов; ***Q*** – число зарегистрированных отказов |
| Н0501 | Оценка по среднему времени восстановления***Q*** | ОБЕСП-Е ОБРАБОТКИ ЗАД ОБЪЕМА ИНФ  Измерительный  +  Расчетный | где:– среднее время восстановления:  ***N*** – число восстановлений;  – время восстановления после ***i***-го отказа;  – допустимое среднее время восстановления |
| Н0502 | Оценка по продолжительности преобразования ***i***-го входного набора данных в выходной | Измерительный  +  Расчетный | где: – допустимое время преобразования ***i***-го входного набора данных;  – фактическая продолжительность преобразования ***i***-го входного набора данных |

### **Оценка надежности ПС в соответствии с ГОСТ 28195-99.**

*Оценка надежности программного средства проводится в следующей последовательности:*

1). На фазе анализа проводится выбор показателей и их базовых значений.

2). Для показателей надежности на всех уровнях принимается единая шкала оценки (от 0 до 1).

3). В процессе оценки надежности на каждом уровне (кроме уровня оценочных элементов) проводится вычисление двух величин:

* абсолютного показателя Pij;
* относительного показателя Rij

где ***j*** – порядковый номер показателя данного уровня для ***i***-го показателя вышестоящего уровня.

Относительныйпоказатель является функцией показателяи его базового значенияи определяется по формуле



*ГОСТ 28195–99* содержит таблицу, содержащую базовые значения для показателей надежности второго уровня (критериев). Данные значения определяются подклассом ПС.

Базовые значения для надежности и ее показателей 3-го уровня формируются методом экспертного опроса с учетом назначения ПС или на основании показателей существующих аналогов или расчетного эталонного ПС. Значения базовых показателей ПС должны соответствовать значениям показателей, отражающих современный уровень качества и прогнозируемый мировой уровень.

4). Каждый показатель надежности второго и третьего уровней характеризуется двумя параметрами:

* количественным значением;
* весовым коэффициентом Vij.

Сумма весовых коэффициентов всех показателей некоторого уровня, относящихся к показателю вышестоящего уровня, постоянна и равна 1:



где: ***J*** – общее количество всех показателей ***j***-го уровня, относящихся к ***i***-му показателю вышестоящего уровня, определенных в стандарте.

*ГОСТ 28195–99* содержит таблицы, содержащие перечни весовых коэффициентов для показателей второго и третьего уровней (критериев и метрик). Количественные величины весовых коэффициентов зависят от фазы жизненного цикла ПС и подкласса ПС.

5). Определениеусредненной оценки **mkq** оценочного элемента по нескольким его значениям (измерениям) mqt осуществляется по следующей формуле (*формула для вычисления значений показателей надежности 4-го уровня*):



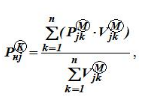
где: ***k*** – порядковый номер метрики; ***q*** – порядковый номер оценочного элемента; ***T*** – число значений (измерений) оценочного элемента; ***t*** – номер значения оценочного элемента.

6)МЕТРИКА Итоговая оценка ***k***-ой метрики ***j***-го критерия определяется по формуле (*формула для вычисления значений показателей надежности 3-го уровня*):



где: – признак метрики; ***Q*** – число оценочных элементов, реально используемых при оценке ***k***-й метрики.

7)КРИТЕРИЙ Абсолютные показатели ***j***-го критерия надежности  вычисляются по формуле (*формула для вычисления значений показателей надежности 2-го уровня*):

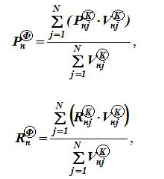


где: ***n*** – число метрик, относящихся к ***j***-му критерию, реально используемых при оценке;  – признак критерия.

8)КРИТЕРИЙ Относительные значения  ***j***-го критерия надежности  по отношению к базовому значению  определяются по формуле



9)ФАКТОР Абсолютные и относительные значения надежности определяются по формулам:



где:  – признак фактора; ***N*** – число критериев надежности, реально используемых при оценке.

### **Модель надежности ПС в соответствии с СТБ ИСО/МЭК 9126-2003.**

Стандарт *СТБ ИСО/МЭК 9126–2003* регламентирует метод оценки качества ПС, основанный на *трехуровневой иерархической модели качества*. На первом уровне модели находятся шесть характеристик качества. Второй уровень составляют подхарактеристики, и третий – метрики качества.

Одной из шести характеристик качества ПС является ***Надежность***.

***Надежность (Reliability)*** – способность программного продукта поддерживать заданный уровень качества функционирования при его использовании в заданных условиях в течение заданного интервала времени.

***Надежность*** включает три подхарактеристики. Это *Стабильность, Устойчивость к ошибке, Восстанавливаемость,*

***Стабильность (Maturity)*** – способность программного продукта избегать отказов вследствие ошибок в программах.

***Устойчивость к ошибке (Fault tolerance)*** – способность программного продукта поддерживать заданный уровень качества функционирования в случаях ошибок в программах или нарушения заданного интерфейса ПП.

***Восстанавливаемость (Recoverability)***– способность программного продукта восстанавливать заданный уровень качества функционирования и данные, поврежденные в случае отказа. Одним из показателей восстанавливаемости является длительность восстановления.

Следует отметить, что в данном стандарте регламентирован только первый уровень модели – уровень характеристик. Приведенные подхарактеристики носят рекомендательный характер. Примеры метрик в стандарте *СТБ ИСО/МЭК 9126–2003* вообще отсутствуют.

С учетом того, что одной из шести характеристик качества ПС является надежность, рассмотрим метод оценки качества, регламентированный в *СТБ ИСО/МЭК 9126–2003,* в применении к надежности ПС.

### **Модель надежности ПС в соответствии с ISO/IEC 25010-2011.**

В стандарте*ISO/IEC 25010:2011* определены две части модели качества. Это модель качества продукта и модель качества в использовании. Данные модели имеют трехуровневую иерархическую структуру (уровни характеристик, подхарактеристик и метрик). Уровни характеристик и подхарактеристик являются регламентированными.

Характеристика надежность является одной из восьми характеристик качества, регламентированных моделью качества продукта.

Модель качества продукта имеет отношение к статическим свойствам программного средства и динамическим свойствам компьютерной системы. Статические свойства программного средства определяются с помощью внутренних метрик (мер) качества. Динамические свойства компьютерной системы и входящих в ее состав ПС определяются с помощью внешних метрик качества.

В соответствии с *ISO/IEC 25010:2011* ***Надежность (Reliability)*** определяется как степень выполнения системой, продуктом или компонентом заданных функций в заданных условиях в течение заданного периода времени. Ограничения надежности в процессе эксплуатации вызваны ошибками в требованиях, проектировании и кодировании.

Подхарактеристиками *Надежности* являются *Завершенность, Готовность, Устойчивость к ошибке, Восстанавливаемость*

***Завершенность (стабильность***, ***Maturity)*** – степень соответствия системы, продукта или компонента потребностям в надежности при нормальной эксплуатации. Завершенность зависит от количества ошибок, оставшихся в системе, продукте или компоненте, и определяет возможность их безотказной работы.

***Готовность (Availability)*** – степень работоспособности и доступности системы, продукта или компонента тогда, когда требуется их использование.

Внешне *Готовность* может быть оценена соотношением времен, в течение которых система, продукт или компонент находится в работоспособном и неработоспособном состоянии. Поэтому Готовность представляет собой комбинацию *Завершенности* (обусловливающей частоту отказов), *Устойчивости к ошибке* и *Восстанавливаемости* (определяющей продолжительность времени восстановления после каждого отказа).

***Устойчивость к ошибке (Fault tolerance)*** – степень функционирования системы, продукта или компонента в соответствии с предназначением, несмотря на наличие сбоев аппаратного обеспечения или ошибок в программном обеспечении.

***Восстанавливаемость (Recoverability)*** – степень восстановления поврежденных данных и переустановления требуемого состояния системы в случае прерывания или отказа продукта или системы.

В случае отказа компьютерная система будет некоторое время находиться в неработоспособном состоянии. Продолжительность этого времени определяется ее восстанавливаемостью.

### **Процесс оценки надежности ПС в соответствии с СТБ ИСО/МЭК 9126-2003.**

Модель процесса оценки надежности, положенная в основу рассматриваемого метода, приведена на рис. 6.4. Данная модель отражает основные стадии и этапы, требуемые для оценки надежности ПС.

Процесс оценки состоит из *трех стадий*: определение требований к надежности ПС, подготовка к оцениванию и процедура оценивания. Данный процесс может применяться после любой подходящей работы жизненного цикла для каждого компонента программного продукта.

***1. Определение требований к надежности***

Целью данной стадии является установка требований в терминах подхарактеристик надежности. Требования выражают потребности внешнего окружения ПС и должны быть определены до начала разработки. Так как ПС разделяется на компоненты, то требования для ПС в целом могут отличаться от требований для отдельных компонентов.

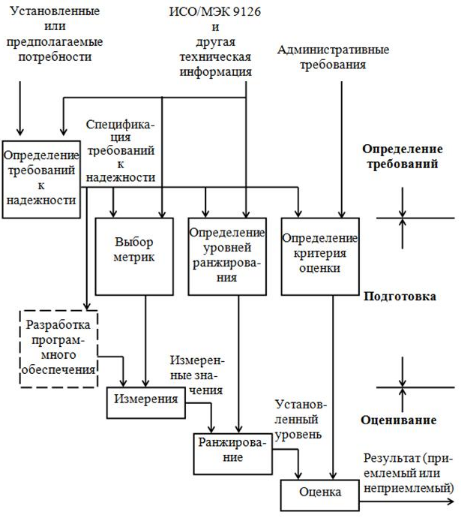


Рис. 6.4. Модель процесса оценки по СТБ ИСО/МЭК 9126–2003

***2. Подготовка к оцениванию***

Целью второй стадии является подготовка основы для оценивания. Данная стадия состоит из трех этапов.

*2.1. Выбор метрик надежности*

С учетом регламентированной в *СТБ ИСО/МЭК 9126–2003* иерархической модели качества уровень характеристики надежность ПС определяется уровнем входящих в нее подхарактеристик, а значения подхарактеристик в свою очередь определяются значениями входящих в них метрик.

В стандарте *СТБ ИСО/МЭК 9126–2003* набор рекомендуемых метрик отсутствует. Поэтому существует потребность в установлении метрик, которые соотносятся с подхарактеристиками надежности ПС. Каждый количественный признак и каждое количественно оцениваемое взаимодействие ПС с его окружением, которые соотносятся с надежностью, могут быть приняты в качестве ее метрики.

*2.2. Определение уровней ранжирования*

Для измерения количественных признаков надежности ПС используются метрики. Измеренные значения отображаются на некоторой шкале. Данные значения не показывают уровень удовлетворения требований к надежности ПС. Для этой цели шкалы метрик должны быть разделены на диапазоны, соответствующие различным степеням удовлетворения требований.

Например, возможны следующие диапазоны ранжирования (рис. 6.5):

* разделение шкалы на две категории: неудовлетворительно и удовлетворительно;
* разделение шкалы на четыре категории (отлично, хорошо, удовлетворительно, неудовлетворительно), ограниченные соответственно запланированным уровнем, текущим уровнем для существующего или альтернативного продукта, и уровнем худшего случая.



Рис. 6.5. Варианты ранжирования измеренных значений метрик

Текущий уровень определяется для управления тем, чтобы новая система не становилась хуже по сравнению с существующей. Запланированный уровень определяет уровень, который считается достижимым при доступных ресурсах. Уровень худшего случая определяет границу принятия пользователем в случае, если изделие не удовлетворяет запланированному уровню (см. рис. 6.5). Так как надежность ПС связана с конкретными потребностями, общие уровни ранжирования невозможны и должны определяться для каждого конкретного оценивания.

*2.3. Определение критерия оценки*

Для определения общей надежности ПС должна быть учтена вся совокупность результатов оценивания различных метрик. Оценщик должен подготовить для этого процедуры, используя, например, таблицы решений или средние взвешенные значения. Обычно при этом учитываются и другие аспекты, такие как время и стоимость, которые являются косвенными факторами надежности ПС.

***3. Процедура оценивания***

Последняя стадия модели процесса оценивания реализуется тремя этапами: «Измерение», «Ранжирование» и «Оценка».

*3.1. Измерение*

Для измерения выбранные метрики применяются к ПС. Результатом являются значения в масштабах метрик.

*3.2. Ранжирование*

На этапе ранжирования устанавливается уровень ранжирования для измеренного значения (см. рис. 6.5).

*3.3. Оценка*

Оценка является последним этапом процесса оценивания ПС, на котором обобщается множество установленных уровней. Результатом является заключение о надежности ПС (приемлемый или неприемлемый уровень надежности).

### **Внутренние метрики надежности ПС.**

В стандарте*ISO/IEC 25010:2011* определены две части модели качества. Это модель качества продукта и модель качества в использовании.

Модель качества продукта имеет отношение к статическим свойствам программного средства и динамическим свойствам компьютерной системы. Статические свойства программного средства определяются с помощью внутренних метрик (мер) качества. Динамические свойства компьютерной системы и входящих в ее состав ПС определяются с помощью внешних метрик качества.

***Внутренние метрики*** – это метрики, измеряющие собственные свойства ПС. Они измеряются в процессе разработки ПС на основе спецификации требований, результатов проектирования, исходного кода или другой документации ПС. Внутренние метрики дают возможность оценить качество промежуточных программных продуктов разработки, предсказывая качество конечного программного средства.

### **Внешние метрики надежности ПС.**

***Внешние метрики*** – это метрики, предназначенные для измерения качества программного продукта путем измерения поведения системы, частью которой является данный продукт. Внешние метрики могут использоваться в процессе эксплуатации и на стадиях тестирования или испытаний в процессах разработки и сопровождения ПС, когда уже созданы исполнимые коды программного продукта.

### **Введение в тестирование ПО. Понятие тестирования ПО. Что может и не может тестирование ПО. Объекты тестирования ПО.**

В 60-е‑80-е годы ХХ века тестирование ПО декларировалось как важная часть жизненного цикла разработки ПО. Но реально, на практике оно использовалось только в организациях, создающих большие и сложные программные комплексы, в тех областях где отказ ПО мог привести к большим финансовым потерям и(или) к гибели людей( военная промышленность, космонавтика, медицина, финансы и т.д.).

Впервые о тестировании ПО по-настоящему заговорили в США в 90-е годы ХХ века. В это время появились мощные персональные компьютеры, развитые операционные системы и средства программирования, которые позволяли решать всё более сложные задачи. Рост мощности компьютерных ресурсов, широкое использование интернета и новых технологий разработки ПО повлекли за собой создание огромных и сложных программных систем. Рост сложности ПО объективно привёл к росту ненадёжности ПО, т.е. к росту количества ошибок в ПО и росту потенциальных проблем. Основным методом обеспечения надёжности ПО являлось тестирование. Поэтому практически во всех организациях разрабатывающих ПО на повестку дня стало реальное тестирование , так как пробиться на рынке, пытаясь продавать некачественное ПО, было невозможно.

Существуют различные подходы определения тестирования ПО. Одни специалисты под тестированием понимают метод проверки корректности (правильности) ПО, где под корректностью понимается степень соответствия ПО своей спецификации. Другие специалисты считают, что тестирование не позволяет установить корректность ПО, т.к. тестирование может служить для доказательства наличия ошибок, но не доказать их отсутствия. Майерс считает, что тестирование –процесс исполнения программ с целью обнаружения ошибок [13, 21, 24, 25, 28].

Одно из широко используемых определений тестирования: ***тестирование*** – это процесс анализа или эксплуатации ПО с целью выявления дефектов [13, 24].

*Тестирование может:*

- обнаруживать ошибки;

- показывать соответствие функций программы ее назначению, т. е. проверять правильность работы (реализацию) функциональных требований;

- отображать надежность как индикатор качества программы.

*Тестирование не может:*

- показывать отсутствие ошибок.

Существующие стандарты определяют ПО как совокупность программ (кодов) и программной документации. Поэтому тестировать в общем можно как собственно ПО, так и программную документацию. Таким образом можно тестировать:

- исполняемые коды программ при их непосредственном запуске и исполнении;

- исходные коды программ без их запуска и исполнения;

- требования к ПО;

- план проекта;

- тестовый план;

- документацию описывающею программную архитектуру;

- тестовые сценарии;

- документацию для конечных пользователей;

- и т.п.

Практически можно тестировать любой промежуточный продукт разрабатываемого программного средства. В дальнейшем под объектом тестирования будет пониматься исполняемое ПО и иные случаи будут оговариваться особо.

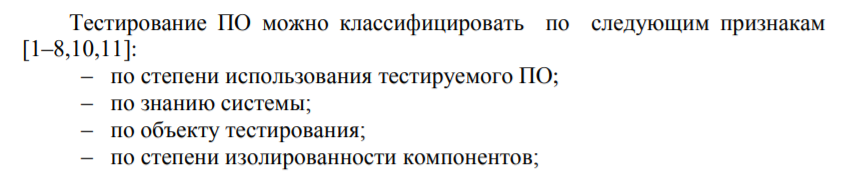
Программы как объекты тестирования имеют две основные *особенности*:

1). Отсутствие полностью определённого эталона, которому должны соответствовать все результаты проверяемого программного обеспечения. Для тестирования программ в качестве эталонов чаще всего используются косвенные данные, которые не полностью отражают функции тестируемой программы.

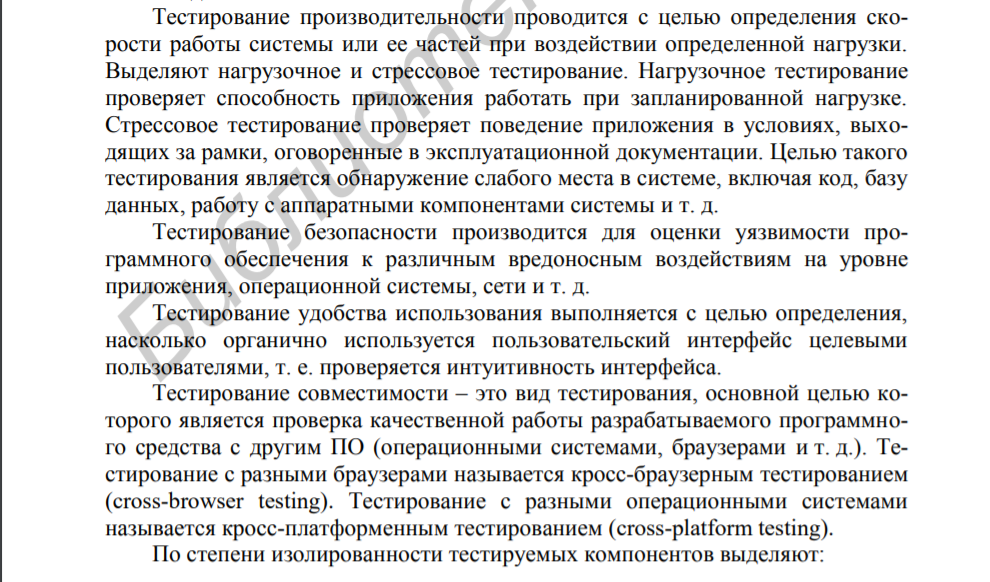
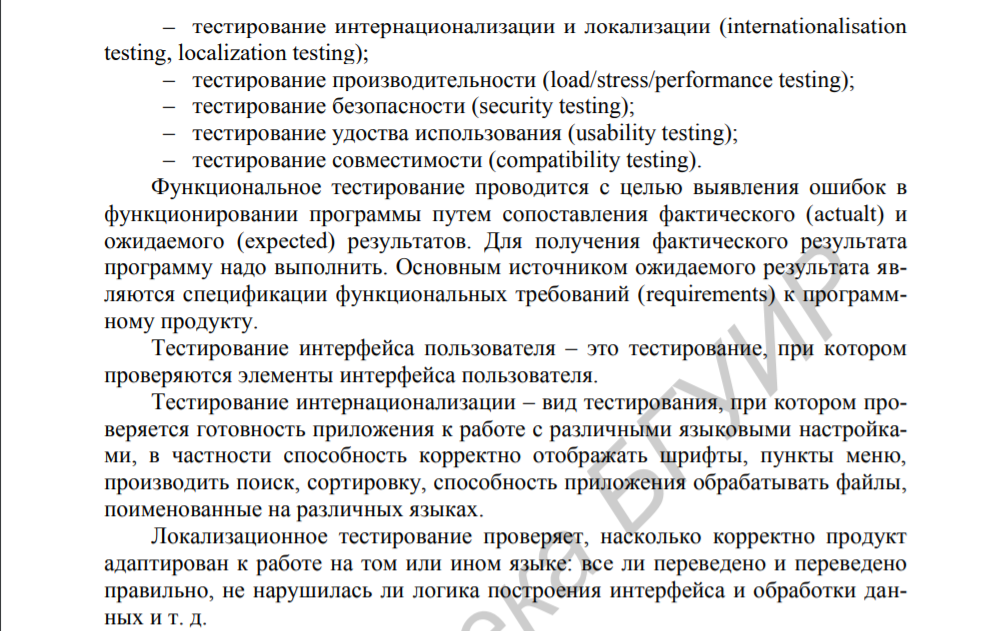
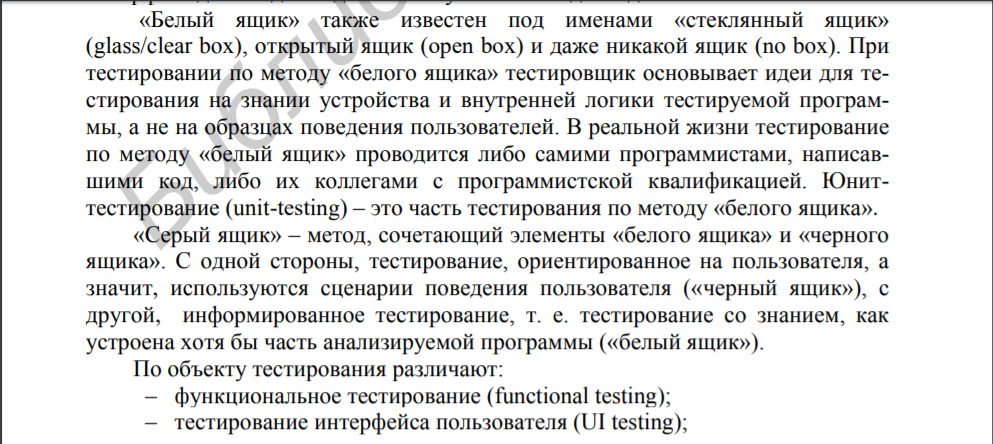
2). Высокая сложность ПО и принципиальная невозможность построения тестовых наборов, достаточных для исчерпывающего тестирования программ.

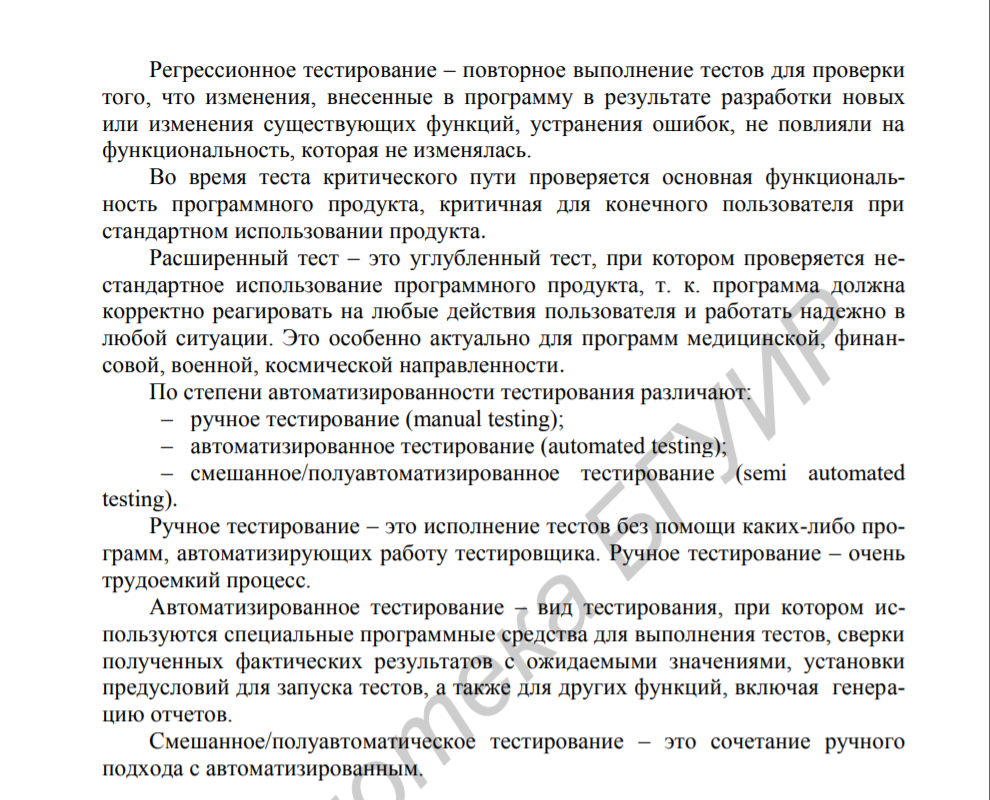
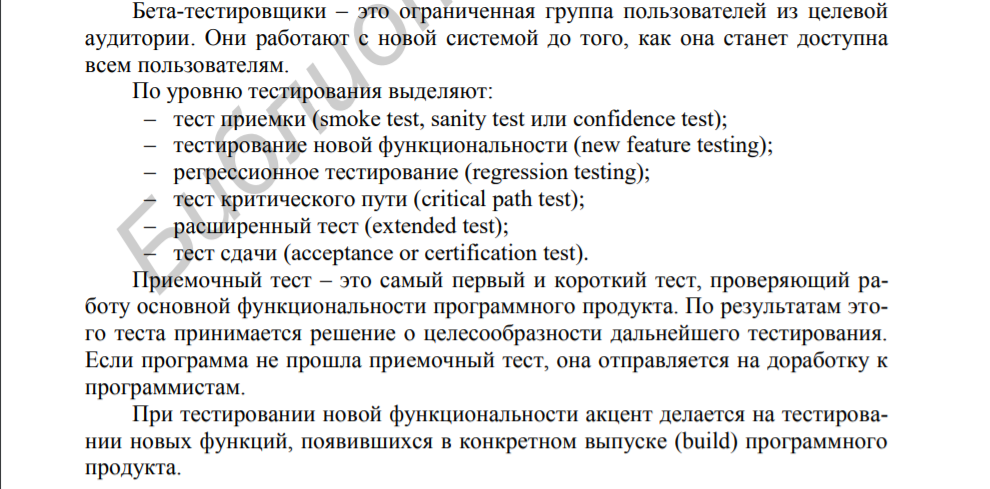
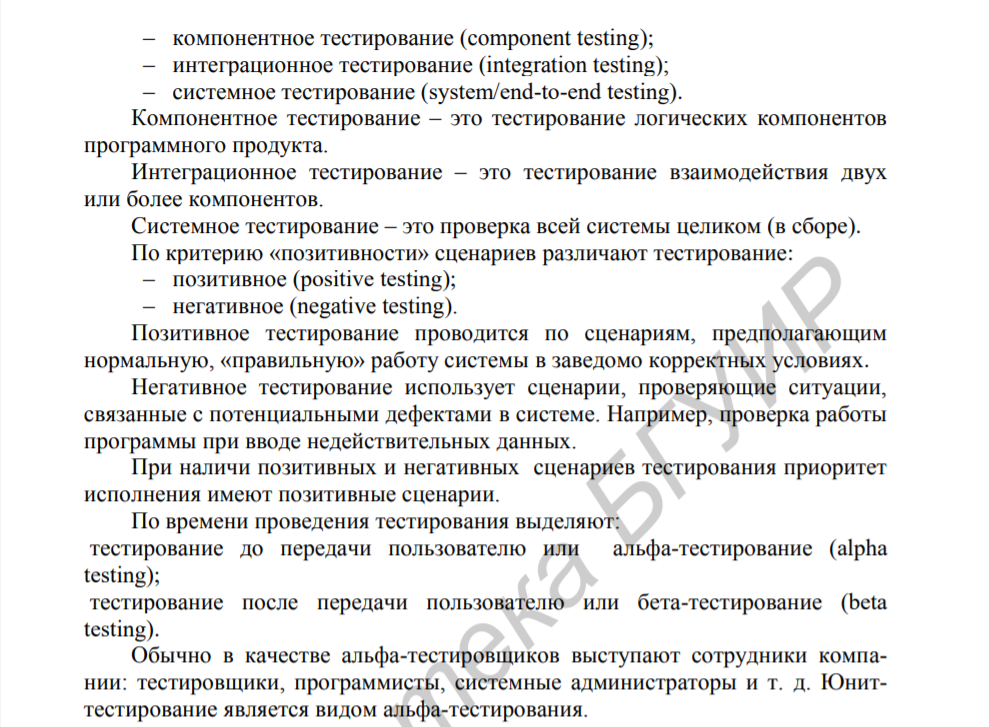
### **--------------------------------------------------------------------------------------------------**

### **Виды тестирования ПО.**









### **Принципы разработки тестов.**

Возможен целый спектр подходов и принципов к разработке тестов (рис. 8.1) [28, 29].



Рис. 8.1. Принципы разработки тестов

Из анализа рисунка видно, что возможны два крайних подхода к процессу разработки тестов.

**Первый подход.**

Данный подход соответствует левой границе спектра (см. рис. 8.1), и заключается в том, что ПО рассматривается как «*черный ящик»* (рис. 8.2), когда в процессе проектирования тестов структура ПО не интересует тестировщика. Данный метод называют методом «чёрного ящика» (black boxtesting method).



Рис. 8.2. Тестирование по принципу “Черный ящик”

Методы, которые рассматривают ПО как «черный ящик», называются *функциональными методами*. В данном случае известны функции, которые должно выполнять ПО. В идеале должна тестироваться работа каждой такой функции по всей области ее определения. Основное место применения тестов «черного ящика» - проверка правильности выполнения функций ПО на уровне интерфейса.

В общем случае тестировщика не интересует прохождение всех путей или ветвей программы. Его удовлетворяет тот факт, что ПО ведет себя так, как указано в спецификации.

В общем случае данный подход практически не осуществим, т.к. проверка работы каждой функции для всей области ее определения практически неосуществима. Поэтому тестирование функций осуществляется с помощью специальных методов за счёт выбора отдельных значений тестовых данных.

Достоинства метода «чёрного ящика»:

* тестирование выполняется тестировщиками, а не программистами и поэтому оно достаточно объективно;
* тестировщику не обязательно знать программный код тестируемого ПО;
* тестирование выполняется с точки зрения конечного пользователя, что важно для эксплуатации ПО пользователем;
* тест-кейсы могут разрабатываться сразу после разработки спецификации требований и до начала кодирования;

**Второй подход.**

Данный подход заключается в изучении структуры ПО и ее логики (рис 8.3). Данный метод называют методом «белого ящика» (white box testing method).

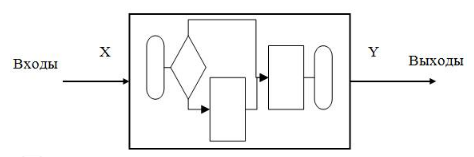


Рис. 8.3. Тестирование по принципу “Белый ящик”

Методы, которые рассматривают ПО как «белый ящик», называют структурными методами.

Объектом тестирования в данном случаи является не внешнее, а внутреннее поведение ПО. Чаще всего анализируются управляющие связи элементов структуры.

Например, была ли выполнена каждая ветвь программы, либо каждый переход условного оператора был выполнен, либо был выполнен каждый путь программы, либо каждый оператор был выполнен хотя бы один раз. Тестирование по данному принципу характеризуется степенью с которой тесты покрывают исходный текст программы. При использовании второго подхода исчерпывающее тестирование по критерию путей для всех входных данных также практически неосуществимо.

Анализ данных методов показывает, что исчерпывающее тестирование практически не возможно и представляет собой в значительной степени экономическую проблему. Поэтому на практике ограничиваются чем то меньшим , то есть так проектируют тесты, чтобы они обеспечивали максимальную отдачу при ограниченных ресурсах (время, средства, люди).

Достоинством метода «белого ящика» является то, что метод «чёрного ящика» может скрыть проблемы, которые метод «белого ящика» отображает. Так, метод «чёрного ящика» может не сообщить о неправильном функционировании объекта, потому что проблемы в работе оказались незаметны. Метод «белого ящика» может обнаружить некорректный объект обеспечив прохождение вычислительного процесса по нужному пути исполнения кода за счёт выбора нужных тестовых данных.

Существует также метод «серого ящика» (gray box testing method), сочетающий в себе нечто среднее между методами «белого ящика» и «чёрного ящика».

### **Информационные потоки процесса тестирования ПО. Аксиомы тестирования ПО.**

На входе процесса тестирования находятся потоки (рис. 8.4) [32]:

· тестируемая программа;

· исходные данные (тесты);

· ожидаемые результаты.

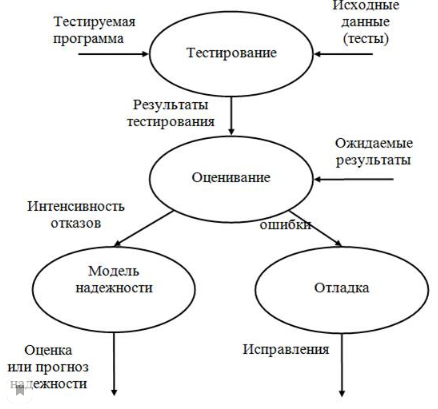


Рис 8.4. Информационные потоки процесса тестирования

Тесты выполняются, и все полученные результаты оцениваются, то есть реальные результаты сравниваются с ожидаемыми результатами.

Когда наблюдается несовпадение с ожидаемым результатом, то фиксируется отказ и начинается отладка, которая должна привести к исправлению ошибки, приведшей к данному отказу. Кроме того, фиксируется статистика о фактах отказов, которая позволяет оценить или спрогнозировать надежность разрабатываемого ПО.

Принципы организации тестирования

Майерс Г. выделил следующие *принципы тестирования*, которые позже были расширены и дополнены [28, 29].

1). Хорошим является то тест, у которого большая вероятность обнаружения ошибки, а не тот, который демонстрирует правильную работу ПО.

2). Нецелесообразно тестировать ПО своей собственной разработки, т.к. тестирование должно быть разрушительным процессом. Тестирование по возможности всегда должна выполнять внешняя группа(еще лучше внешняя организация), которая стоит отдельно от программистов и проекта.

3). Необходимо готовить тесты как для правильных (в соответствии спецификацией), так и не правильных исходных данных.

4). Одна из сложных проблем при тестировании – решить, когда нужно закончить тестирование. Необходимо разработать критерии, позволяющие обоснованно принять решение об окончании тестирования.

5). Необходимо детально изучать результаты каждого теста, чтобы не пропустить ошибку.

6). Любое тестирование должно быть воспроизводимым. Для чего тесты необходимо документировать и хранить в виде удобном для повторного использования.

7). Необходимой частью всего теста является заранее подготовленное описание выходных результатов. Лучше всего разрабатывать самопроверяющиеся тесты использующие, например, либо контрольное соотношение, либо попадание в заданную область, позволяющие автоматически сверять ожидаемые результаты с фактическими.

8). Нельзя изменять программу, чтобы облегчить ее тестирование. Программа должна тестироваться «как есть».

9). Тестирование, как почти всякая другая деятельность, должно начинаться с постановки целей. Как уже неоднократно отмечалось, тесты должны быть спроектированы, реализованы, проверены и, наконец, выполнены.

### **Структурное тестирование ПО. Понятие потокового графа, пути, базового пути, ветви, цикломатической сложности.**

Все методы тестирования ПО делятся на [13, 24, 25, 32]:

1) методы структурного тестирования ПО;

2) методы функционального тестирования ПО.

Структурное тестирование используется на ранних этапах тестирования, то есть на уровне тестирования модулей (7-я работа процесса разработки, см. подразд. 4.1). Функциональное тестирование используется практически во всех работах процесса разработки, связанных с тестированием (работы 7, 8, 9, 11 процесса разработки).

Структурное тестирование основано на понятиях [13, 24, 25, 32]:

1) потокового графа;

2) пути;

3) цикломатической сложности ПО.

***Потоковый граф*** представляет собой структурную модель, показывающую связь между отдельными его элементами.

В общем случае граф

***G = (V, E)***

состоит из множества вершин ***V*** и множества дуг ***E:***



Дуги отображают потоки управления в программе (модуле), то есть передачи управления между операторами. Дугу можно представить в виде упорядоченной пары вершин



или записать в таком виде

Различают ***операторные*** и ***предикатные вершины***. Из операторной вершины выходит только одна дуга, из предикатной – две.

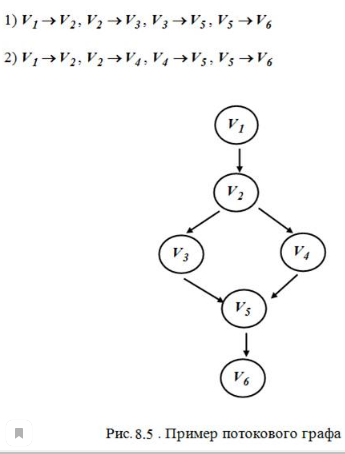
Предикатные вершины соответствуют простым условиям (аналог – условный оператор (оператор ветвления) или оператор цикла в языках программирования). Операторной вершине соответствует или линейный участок программы или один оператор с одним входом и одним выходом.

На рис. 8.5 приведен пример потокового графа.

На данном рисунке ***V2*** - предикатная вершина; ***V3***, ***V4***, ***V5*** - операторные вершины

Если представлять некоторую программу (модуль) в виде потокового графа, то под ***путем*** в графе понимается последовательность вершин для которой существуют дуги ***V1 →Vk , Vk → Vl , …, Vi → Vj , …, Vm → Vn*** . Путь начинается в вершине ***V1***, проходит через вершины ***Vk*** , ***Vl*** ,**…, *Vi , Vj , …, Vm*** изаканчивается вершиной ***Vn*** . ***V1*** соответствует первому выполняемому оператору программы (модуля) и называется начальной вершиной графа. ***Vn*** соответствует последнему выполненному оператору программы (модуля) и называется конечной вершиной графа.

В нашем примере (см. рис. 8.5) возможны пути, соответствующие следующим последовательностям дуг:



Под ***ветвью*** графа понимается часть пути, начинающаяся от начальной или предикатной вершины графа и заканчивающаяся предикатной или конечной вершиной графа.

Для графа, приведенного на рис. 8.5, можно выделить следующие ветви:

***1)*** ***1 – 2***

***2)*** ***2 – 3 – 5 – 6***

***3)*** ***2 – 4 – 5 – 6***

Одной из мер сложности ПО является *цикломатическое число*.

***Цикломатическое число*** ***(число МакКейба)*** определяет структурную сложность графа по сравнению с линейным графом и позволяет оценить трудоемкость тестирования программы (модуля).

***Цикломатическую сложность (цикломатическое число***) модуля можно определить *следующими способами* [32]:

1) *Определение через количество вершин и количество связывающих вершины дуг графа*.

***Z(G)=E-V+2=6-6+2=2.***

В данном выражении ***Е*** – число дуг графа, ***V*** – число вершин графа. Для рис. 8.5 ***Е = 6***, ***V = 6***.

В выражение ***Z(G), Z*** обозначает цикломатическое число, а ***G*** обозначает, что сложность является функцией графа.

*2)* *Определение через количество регионов графа.*

***Z(G)=R=1+1=2,***

где ***R*** представляет собой число регионов графа.

***Регионы*** представляют замкнутые области на графе, образованные дугами и вершинами. Окружающая граф среда является дополнительным регионом. Например, для примера, приведенного на рис. 8.5, имеется один регион, образованный вершинами ***V2*** , ***V3*** , ***V4*** , ***V5*** и связывающими их дугами. Дополнительным регионом является окружающая среда графа. Таким образом, общее количество регионов данного графа равно 2.

3) *Определение через количество предикатных вершин графа*.

Для предикатных вершин, из которых выходит две дуги, цикломатическая сложность

***Z(G)=P+1***,

где ***Р*** – число предикатных вершин.

В нашем примере (рис. 8.5) ***Р = 1***. Поэтому

***Z(G)=1 + 1 = 2.***

Цикломатическое число определяет:

* Количество базовых независимых путей в модуле.
* Количество тестов, гарантирующее полное покрытие всех ветвей графа модуля.

***Независимым*** называется путь, который вводит новый оператор обработки или новое условие, то есть независимый путь содержит дугу, не входящую в ранее определенные пути.

Все независимые пути графа образуют базовое множество. Если в пути встречаются операторы множественного выбора, например switch (C/C++), то необходимо обеспечить переход в каждом направлении.

Например, конструкция оператора switch имеет следующий вид:

switch (a)

{

case ‘1’ : operator1; break;

case ‘2’ : operator2; break;

case ‘3’ : operator3; break;

default : operator4;

}

Фрагмент схемы алгоритма данной конструкции приведен на рис. 8.6, соответствующий граф – на рис. 8.7. На рис. 8.6 и рис. 8.7 цифрами 1 – 4 обозначены соответствующие операторы конструкции switch. На рис. 8.7 номера 5 – 7 соответствуют ветвлениям (предикатным вершинам) схемы алгоритма.

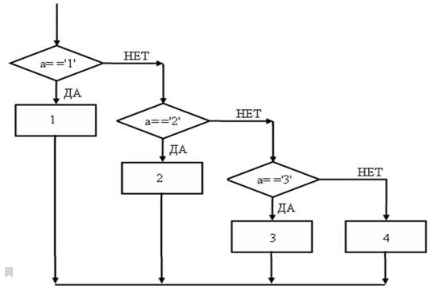


Рис 8.6. Схема алгоритма оператора switch

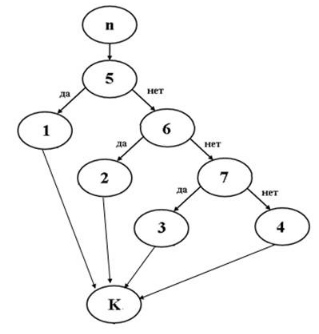


Рис. 8.7. Граф оператора switch

Цикломатическое число равно:

1) ***Z(G) = E – V + 2 = 11 – 9 + 2 = 4;***

2) ***Z(G) = R = 1 + 1 + 1 + 1 = 4***

3) ***Z(G) = P + 1 = 3 + 1 = 4***

Таким образом, цикломатическое число показывает, что для программы (модуля), реализующей фрагмент алгоритма (см. рис. .8.6), необходимо 4 теста с целью покрытия всего графа (см. рис. 8.7).

8.8.2.2-3 пропущено

### **Метод тестирования базовых путей. Тестирование циклов.**

Автор метода - Том МакКейб [32].

Данный метод позволяет:

1) получить оценку сложности модуля;

2) на основе полученной оценки сложности определить необходимое количество тестов для определения набора базовых путей через тестируемый модуль. Полученные тесты гарантируют выполнение каждой ветви модуля или каждого оператора модуля хотя бы один раз.

*Суть метода* заключается в следующем:

1. На основе схемы алгоритма модуля формируется потоковый граф. Пусть схема алгоритма имеет общий вид, приведенный на рис. 8.9. Данной схеме соответствует граф, изображенный на рис. 8.10.

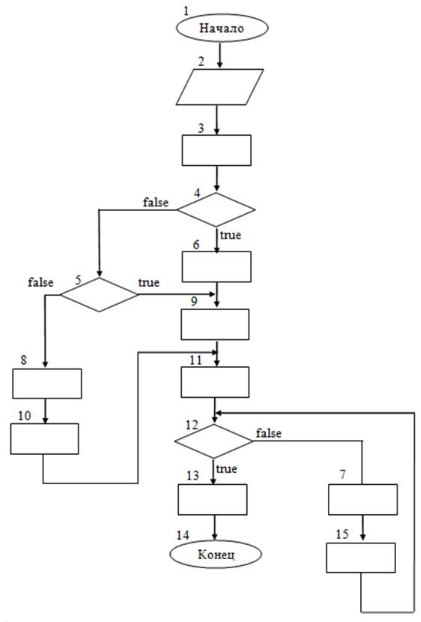


Рис. 8.9. Алгоритм работы модуля

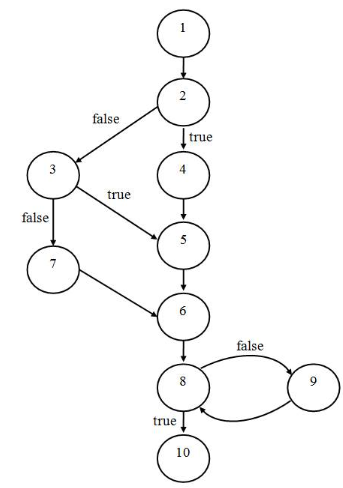


Рис. 8.10. Потоковый граф модуля

2. Определяется цикломатическая сложность потокового графа, с помощью перечисленных в п. 8.8.2.1 формул. Для рассматриваемого примера

***Z(G) = 4.***

3. Определяется базовое множество независимых путей. Для этого в начале выбирается любой один путь от входа в модуль к его выходу. Данный путь назовем *первым базовым путем*.

Для нашего примера первый базовый путь:

***1)*** ***1 – 2 (true) – 4 – 5 – 6 – (8 (true)) – 10.***

Чтобы получить второй путь, необходимо в первом базовом пути поменять результат первого условного выражения, при этом сохраняя по возможности результаты остальных условных выражений. Т.е. после того как произошло воссоединение с первым базовым путем, его следует придерживаться до самого выхода. Любые встреченные условные выражения, которые не являются частью первого базового пути, могут принимать любые значения.

***2) 1 – 2 (false) – 3 (true) – 5 – 6 – 8 (true) – 10.***

Для получения третьего пути необходимо поменять в первом базовом пути уже результат не первого, а второго условного выражения:

***3) 1 – 2 (true) – 4 – 5 – 6 – 8 (false) – 9 – (8 (true)) – 10.***

В третьем базовом пути с целью достижения конечной вершины “10” оператор “8” принял значение “true”. Однако, данное условие не будем считать изменением базового пути. Это вспомогательный элемент базового пути, позволяющий потоку управления дойти до выходной вершины.

Когда все решения первого базового пути пройдены и изменены, необходимо перейти ко второму базовому пути и изменять результаты тех условных выражений, которые не входили в базовый путь.

Преобразуем второй путь. В результате получим:

4) ***1 – 2(false) – 3(false) – 7 – 6 – 8 (true) – 10.***

После того как все операторы в модуле выполнены в каждом направлении, набор тестовых путей будет закончен.

Достоинством метода является его независимость от вида потокового графа, т.е. метод применим как для циклических, так и для ациклических графов.

**Тестирование циклов**

Тестирование циклов производится по принципу “белого ящика”, при этом основное внимание уделяется правильности конструкции программы [32].

Возможны следующие ***типы циклов***:

1. *простые* (рис. 8.11)

2. *вложенные* (рис. 8.12)

3. *неструктурированные* ( go to на рис. 8.13).

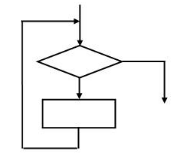


Рис. 8.11. Простой цикл

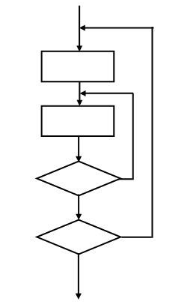


Рис. 8.12. Вложенный цикл

****

Рис. 8.12. Неструктурированный цикл

Для проверки простых циклов с числом повторений ***n*** можно использовать следующий возможный набор тестов:

* ни одного прохода цикла;
* один проход цикла;
* ***m < n*** проходов цикла, где ***m*** – достаточно малое число проходов, если ***n*** достаточно велико;
* ***n*** – 1, ***n***, ***n*** + 1 проходов цикла, если ***n*** достаточно мало.

Если цикл простой и число повторений не определено, то должны быть тесты с однократным выполнением тела цикла и без выполнения тела цикла.

В случае вложенных циклов общее число путей может быть достаточно большим, поэтому для сокращения числа тестов применяется следующая *методика*:

1)вначале выбирается самый внутренний цикл и устанавливается минимальное значение параметров для всех охватывающих циклов;

2)для самого внутреннего цикла проводятся тесты простого цикла;

3)переходят в следующий по порядку объемлющий цикл и проводят с ним тесты простого цикла, при этом устанавливают минимальные значения параметров всех объемлющих и внутренних циклов;

4)шаги методики повторяют до тех пор, пока не будут протестированы все циклы программы.

Неструктурированные циклы тестировать нельзя и они должны быть приведены к структурированному виду.

### **Функциональное тестирование ПО. Общие сведения. Цели функционального тестирования. Уровни функционального тестирования.**

Функциональное тестирование является основным видом тестирования ПО. Каждая функция программы тестируется и при этом делается вывод об ее правильности. Очевидно, что по всей области определения проверить функцию невозможно и поэтому каждая функция проверяется на правильность в некоторых точках области её определения [13, 28, 32].

Функциональное тестирование не является альтернативой структурному тестированию – это дополняющий подход, позволяющий обнаружить другие классы ошибок. Например:

· обнаружение некорректных или отсутствующих функций;

· обнаружение ошибок интерфейса;

· обнаружение ошибок во внешних структурах данных (файлы, базы данных).

В отличие от структурного, функциональное тестирование используется на более поздних этапах тестирования.

В зависимости от того используются или нет средства автоматизации тестирования функциональное тестирование подразделяется на:

1) ручное тестирование;

2) автоматическое или автоматизированное тестирование(automated testing).

Ручное тестирование подразумевает выполнение тестов вручную. Автоматическое или автоматизированное тестирование подразумевает привлечение инструментальных средств для автоматизации тестирования.

Целями функционального тестирования являются:

1) поиск тестировщиками в тестируемом ПО ошибок и , чем больше будет их найдено и серьёзнее они будут, тем лучше. При этом очевидно, что даже для небольших программ провести 100% исчерпывающее тестирование невозможно в силу ограниченных компьютерных, материальных и временных ресурсов. Поэтому на практике тестирования всегда недостаточно, и в тестируемом ПО остаются не выявленные ошибки;

2) документирование тестировщиками найденных ошибок с целью дальнейшего их исправления программистом. При этом необходимо, чтобы абсолютно каждая найденная ошибка была задокументирована;

3) определение соответствия тестируемого ПО предъявляемым к нему требованиям и принятие объективного заключения о возможности поставки протестированного ПО заказчику. Причём это заключение, как правило, делает менеджер проекта. При этом тестировщик путём предоставления максимально полной и объективной информации о качестве тестируемого ПО влияет на принятие данного заключения.

Функциональное тестирование можно классифицировать а зависимости от объёма проводимого тестирования. При этом различают три уровня функционального тестирования:

1) приёмочный тест (smoke test);

2) критический тест (critical path test);

3) расширенный тест (extended test).

Приёмочный тест является самым первым и коротким тестом, проверяющим работу основной функциональности тестируемого ПО. В зависимости от сложности ПО он длится от получаса до 2-3-х часов максимум, и затем принимается решение о целесообразности дальнейшего тестирования. При этом, если ПО не прошло приёмочный тест, то оно отправляется на доработку программистам, так как нет смысла выполнять дальнейшее тестирование.

Критический тест является основным по объёму тестирования видом теста. Во время критического теста проверяется основная функциональность тестируемого ПО, критичная для конечного пользователя при стандартном использовании ПО. В рамках данного тестирования проверяется большинство основных требований, предъявляемых к тестируемому ПО.

Расширенный тест представляет собой углубленный тест, при котором проверяется нестандартное использование тестируемого ПО. При этом прогоняются различные сложные, логически запутанные сценарии и совершаются действия, которые пользователь будет совершать редко. Расширенный тест используется в тех предметных областях, где риски велики и где отказ ПО может привести к большим финансовым потерям и(или) гибели людей. Протестированное ПО при этом должно работать надёжно в любой ситуации. Для тех предметных областей, где риски не велики, ПО может быть и не столь надёжным, и поэтому нет смысла тратить временные и материальные ресурсы на разработку и прогон сложных и запутанных тестовых сценариев.

Наиболее известными функциональными методами тестирования ПО являются:

1. Метод эквивалентного разбиения;

2. Метод анализа граничных условий.

### **Понятие класса эквивалентности. Примеры.**

Разбиение на классы эквивалентности (КЭ) представляет собой технологию проектирования тестов, ориентированную на снижение числа тестов необходимых для подтверждения корректности функциональных возможностей ПО. Основная идея, стоящая за разбиением на классы эквивалентности, заключается в том, чтобы разбить область ввода ПО на классы данных. При этом, очевидно, если проектировать тесты для каждого класса данных, но не для каждого элемента класса, то общее количество требуемых тестов уменьшится.

Принадлежность двух элементов данных к одному и тому же классу эквивалентности приводит к выбору общего набора операторов в тестируемом модуле.

Принадлежность двух элементов данных к различным классам эквивалентности означает, что существует по крайней мере одна строка кода, требуемая для обработки одного элемента данных, которая не будет использоваться при обработке другого элемента данных.

Таким образом, если от выполнения двух тестов ожидается один и тот же результат, то с точки зрения надёжности ПО эти тесты считаются эквивалентными.

В общем группа тестов представляет собой класс эквивалентности, если выполняются следующие условия:

1) Все тесты предназначены для выявления одной и той же ошибки;

2) Если один из тестов выявляет ошибку, то остальные тесты скорее всего выявят эту же ошибку (с высокой вероятностью);

3) Если один из тестов не выявит ошибку, то и остальные тесты скорее всего не выявят эту ошибку (с высокой вероятностью).

Классы эквивалентности выделяются путем выбора каждого входного условия. Обычно классов эквивалентности много, поэтому лучше их представлять в структуры вида таблица (таблица 8.1). Символ • в таблице 8.1 обозначает класс эквивалентности.

Таблица 8.1.

Примеры классов эквивалентности

|  |  |  |  |
| --- | --- | --- | --- |
| Входное или выходное условие (событие) | Правильный КЭ (ПКЭ) | | Неправильный КЭ (НКЭ) |
| 1) Ввод числа от 1 до 99 (необходимо протестировать поле, принимающее только числовые значения в диапазоне от 1 до 99) | · X принадлежит [1..99] | | · Х > 99  · отрицательные числа  · буквы и другие символы |
| 2) Рисование прямой | · от одной точки (пиксела) до 5 сантиметров длиной | | · отсутствие рисунка  · линия длиннее 5 сантиметров  · линия не является прямой |
| 3) Программа задаёт вопрос: “Вы уверены?” (Да/Нет) и поэтому необходимо ответить | · {Y, Д, y, д}  · {N, n, Н, н} | | · X не принадлежит множеству {Д, Н, д, н, Y, y, N, n} |
| 4) Ввод символа большой буквы английского алфавита | · X принадлежит [‘A’ ..’Z’] | | · X > ‘Z’  · X <’A’ |
| 5) Если условие ввода задаёт конкретное значение *a* | · X=a | | · X>a  · X<a |
| 6) Описывает число значений, например, “В автомобиле может ехать от одного до 5 человек” | · X принадлежит [1..5] человек | · ни одного человека  · более пяти человек (X>5) | |
| 7) Надо протестировать раздел входа в сайт (поля для ввода login и password) | · Login = X  · Password = Y  где X, Y – истинные значения | · login ≠ x, password ≠y;  · login = x, password ≠ y;  · login ≠ x, password = y;  · login = ‘ˍ’, password = ‘ˍ’;  · login = x, password = ‘ˍ’;  · login = ‘ˍ’, password = y. | |
|  |  |  |  |

**Пример разбиения на КЭ**

Рассмотрим программный компонент, вычисляющий стоимость перевозки некоторого продукта с известным весом по заданному почтовому индексу получателя. В программе имеется GUI-интерфейс, через который можно вводить почтовый индекс и другие данные, например, вес (масса брутто) пересылаемого продукта.

Входные данные:

‑ Почтовый индекс

‑ Вес продукта

Выходные данные:

‑ стоимость перевозки

Простой подсчет допустимых и недопустимых значений почтовых индексов и вложенных значений массы показывает, что практически исчерпывающее тестирование данной программы невозможно. Поэтому необходимо уменьшить количество тестов используя, например, метод разбиения области входных данных на классы эквивалентности (КЭ).

В общем, поиск КЭ – процесс субъективный. Два тестировщика, анализирующих одну и ту же программу составят различные КЭ. При этом, надо выявить как можно больше КЭ, так как это сделает тестирование более эффективным.

Областью ввода этой программы являются:

1) Шестиразрядный почтовый индекс;

2) Масса брутто отправляемого груза (1-10) кг.

Область ввода почтового индекса может быть разбита на класс допустимых вводов (ПКЭ) и область недопустимых вводов (НКЭ).

ПКЭ:

· Все шестизначные наборы цифровых символов, образующих рабочий почтовый код.

НКЭ:

· Наборы цифровых символов, содержащие менее шести символов;

· Наборы цифровых символов, содержащие более шести символов;

· Наборы из шести символов, не являющиеся рабочим почтовым кодом;

· Наборы из нецифровых символов;

· Наборы спецсимволов.

Область ввода массы брутто груза также может быть разбита на ПКЭ и НКЭ.

|  |  |
| --- | --- |
|  |  |
|  |  |

Масса брутто

1 10 груза

НКЭ1 ПКЭ НКЭ2

ПКЭ:

· Масса брутто груза в диапазоне (1-10) кг.

НКЭ:

· Все значения меньше 1 кг;

· Все значение больше 10 кг;

· Отрицательные значения;

· Нецифровые значения.

Нужно спроектировать тесты, которые выполняли бы проверку по крайней мере по одному представителю каждого ПКЭ и НКЭ. В случае ввода недопустимых данных должно быть выдано соответствующее сообщение об ошибке, если оно определено в требованиях спецификации. При этом, при вводе недопустимых данных (НКЭ) программа, по меньшей мере, не должна завершиться аварийно, вызывать искажения данных или вести себя непредсказуемым образом.

### **Метод эквивалентного разбиения. Метод анализа граничных значений. Примеры.**

В соответствии с данным методом проектирование тестов проводится в *два этапа* [24, 25, 28, 32].

***На первом этапе*** в соответствии со спецификацией область входных данных делится на классы эквивалентности. Для каждого входного условия из спецификации формируется два или более класса эквивалентности. Это так называемые *ПКЭ (правильный класс эквивалентности)* и *НКЭ (неправильный класс эквивалентности)*. ПКЭ – область допустимых для тестируемой программы значений входных данных. НКЭ – область недопустимых значений входных данных.

Например:

Любой набор данных из класса эквивалентности обладает тем общим свойством, что приводит к выбору одного и того же набора операторов в тестируемом модуле. Таким образом, любые два теста из одного класса эквивалентности с высокой вероятностью могут обнаружить одну и ту же ошибку ПО.

***На втором этапе*** осуществляется собственно проектирование тестов с помощью сформированных классов эквивалентности. Для ПКЭ тесты проектируются таким образом, чтобы каждый тест покрывал как можно больше еще не покрытых ПКЭ. Процесс продолжается до тех пор, пока не будут покрыты все ПКЭ.

Затем проектируются тесты для каждого НКЭ, при этом они проектируются таким образом, чтобы каждый тест покрывал один и только один НКЭ. Процесс продолжается до тех пор, пока не будут покрыты все НКЭ, но только не общими тестами.

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

Например, спецификация устанавливает тип книги при поиске (Вычислительная техника, Программирование, Общий) и количество (1 – 9999).

Тогда тест:

XYZ 0 ,

в котором задан неверный тип книги и количество, отображает два ошибочных условия:

1) неправильный тип книги

2) неправильное количество.

Такой тест вероятно не будет осуществлять проверку количества, так как программа может ответить «XYZ – несуществующий тип книги» и не проверять остальную часть входных данных.

В общем случае выбор тестов из каждого класса эквивалентности можно осуществить детерминированным или случайным способом.

#### **Метод анализа граничных значений**

На практике ошибки ПО часто проявляются на границах области ввода. Под граничными значениями понимаем ситуацию, возникшую на границе определенного спецификацией входного или выходного условия. Суть метода заключается в выполнении следующих правил [24, 25, 28, 32]:

**1-е правило.** Если входное условие, то есть условие ввода, задает диапазон чисел ***X*[*1, 100*],** то необходимо создать тесты:

· для значений ***1*** и ***100***;

· для значения чуть левее ***1*** и чуть правее ***100***.

Например,

***x1 = 1.0;***

***x2 = 100.0;***

***x3 = 0.9;***

***x4 = 100.1.***

**2-е правило.** Если условие ввода задает дискретное множество значений, то необходимо создать тесты:

· для проверки минимального и максимального значений;

· для значения чуть меньше минимума и чуть больше максимума.

Например, если ***X*[*2, 5, 7***], то

***x1 = 2;***

***x2 = 7№***

***x3 = 1;***

***x4 = 8.***

Если задан входной файл, содержащий 1000 записей, то необходимо создать тесты для ввода нуля записей, одной записи, 1000 записей и 1001 записей.

**3-е правило.** Ранее названные правила 1 и 2 применяются к условиям области вывода. Например: выходной (формируемый) файл содержит 1000 записей. Поэтому нужно создать тесты для создания соответственно 0, 1, 1000 и 1001 записи.

**4-е правило.** Если внутренняя структура данных программы является упорядоченным множеством (массив, список), то разрабатываются тесты, проверяющие эти структуры на границах.

Например, если задан на языке ***Си*** массив целых чисел, состоящий из 100 элементов, т.е. ***int a[100]***, то необходимо создать тесты для проверки соответственно ***a[-1], a[0], a[99], a[100]*** элементов массива.

Анализ граничных значений отличается от эквивалентных разбиений двумя моментами:

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 выходные КЭ (пространство результатов).

**Пример тестирования поля даты**

В приложении есть поле, куда можно вводить дату (mm/dd/yy), mm – две цифры месяца, dd – две цифры дня, yy – две цифры года. Начнём с КЭ. Какая дата будет правильной? Фактически у нас четыре переменных. Составим тесты (таблица 8.2):

1) Проверяем минимальные значения (можно проверить сразу три правильных значения (месяц, день, год);

2) Аналогично проверяем максимальные значения (не забыть про високосный год – 29 февраля);

3) Проверяем, как поле реагирует не на цифры;

4) Проверяем пустое значение.

Таблица 8.2.

Примеры классов эквивалентности для поля даты

|  |  |  |
| --- | --- | --- |
| Тест | Правильная граница  (mm/dd/yy) | Неправильная граница  (mm/dd/yy) |
| 1) min day, month, year | 01/01/00 | **00**/01/00  01/**00**/00  1/1/**-1** |
| 2) Max day, month, year      Max day (високосный год) | 12/31/99      02/29/20  02/28/01  04/30/65 | **13**/31/99  12/**32**/99  12/31/**100**  02/**30**/20  02/**29**/01  04/**31**/65 |
| 3) Буквы, неправильные разделители, специальные символы |  | March  03.12.06  !@#$%^&\*()\_+{} |
| 4) Пустое значения |  |  |

### **Документирование тестов. Структура Test Case. Примеры.**

Тесты документируются в виде «тест-кейсов», или «тестовых случаев»(«test-cases») [13].

Gод тест-кейсом понимается набор тестовых входных данных, условий выполнения и ожидаемых результатов, разработанных с целью проверки некоторого пути выполнения программы или проверки соответствия некоторому требованию.

Тест-кейс представляется в виде документа, который может включать следующие поля:

1) идентификатор тест-кейса;

2) связанные с тест-кейсом требования;

3) приоритет тест-кейса;

4) модуль и подмодуль приложения, к которым относится тест-кейс;

5) название тест-кейса;

6) исходные данные, необходимые для тест-кейса;

7) шаги для выполнения тест-кейса;

8) описание ожидаемых результатов по каждому шагу выполнения тест-кейса;

9) поле статуса об успешном или ошибочном выполнении тест-кейса;

10) автор тест-кейса;

11) время последнего выполнения тест-кейса;

12) последний полученный результат;

13)связанный с тест-кейсом дефект(баг), если он есть.

В общем случае отдельные поля шаблона тест-кейса могут опускаться.

***Достоинствами использования тест-кейсов*** является то, что они:

1) дают тестировщикам структурированный системный подход, что в общем снижает вероятность пропуска ошибок;

2) представляют собой один из способов тестирования проектной документации ещё до появления первой промежуточной версии программного продукта;

3) значительно ускоряют процесс регрессионного тестирования;

4) позволяют отслеживать статистику проводимого тестирования;

В таблице 8.3 приведены примеры тест-кейсов «Сложение чисел А и В».

Таблица 8.3.

Примеры тест-кейсов «Сложение целых чисел А и В»

|  |  |  |
| --- | --- | --- |
| Идентификатор  тест-кейса | Название тест-кейса и его описание | Ожидаемый результат |
| 1.1 | Сложение целых чисел А и В.  1. В поле А ввести число 3.  2. В поле В ввести число 4.  3. Нажать кнопку «Сложить».  4. Проверить полученное значение в поле С. | 4. Значение в поле С равно 7. |
| 1.2 | Сложение целых чисел А и В.  1. Проверить, что тестируемая программа корректно суммирует числа А и В. | 1. Суммирует корректно. |
| 1.3 | Сложение целых чисел А и В.  1. В поле А ввести корректное целое число.  2. В поле В ввести корректное целое число.  3. Нажать кнопку «Сложить».  4. Проверить полученное значение в поле С.  5. Повторить шаги 1 - 4 для значений: 0; максимально и минимально допустимых значений; 1.5; символов и спецсимволов. | 4. Значение поля С равно сумме А и В. |

Анализ трёх приведенных тест-кейсов показывает следующее. Первые два тест-кейса плохие. В первом тест-кейсе все детали прописаны до мелочей. При повторных выполнениях данного тест-кейса всегда будут выполняться строго одни и те же действия, что в общем снижает вероятность обнаружения ошибки.

Второй тест-кейс является слишком общим. Его в общем сложно выполнить по многим субъективным и объективным причинам, что может привести к его невыполнению.

Третий тест-кейс является наилучшим из приведенных тест-кейсов, так как в нём:

- отсутствует привязка к конкретным значениям тестовых данных;

- задано выражение, с помощью которого проверяется правильность результатов выполнения тестируемой программы;

- в тест-кейсе перечислены тестовые данные, повышающие эффективность тестирования.

При написании тест-кейсов следует:

1) использовать простой технический стиль;

2) в русском языке использовать безличную форму: «открыть», «закрыть» и т.п. (вместо «откройте»);

3) описывать поведение систем следующим образом: «появляется окно», «приложение закрывается» и т.п.;

4) обязательно указывать точные названия всех элементов приложения;

5) не объяснять базовые понятия работы с операционной системой.

В таблице 8.4 приведен пример тест-кейса «Открытие PDF-файла».

Таблица 8.4.

Пример тест-кейса «Открытие PDF-файла»

|  |  |  |
| --- | --- | --- |
| Идентификатор тест-кейса | Название тест-кейса и его описание | Ожидаемый результат |
| 1.1 | Открытие PDF-файла.  1. Нажать кнопку «Открыть».  2. Выбрать файл из диалогового окна.  3. Нажать кнопку «ОК».  4. Проверить отображение открытого файла. | 1. Появляется диалог выбора файлов.  2. Выбранный файл помечается выделением.  3. Диалог выбора файлов закрывается.  4. Файл отображается в рабочей области приложения. |

### **Методика тестирования ПО. Тестирование модулей.**

В соответствии с *СТБ ИСО/МЭК 12207-2003* процесс разработки программных средств включает 13 работ (см. подраздел 4.1). Из них тестированию соответствуют 7, 8, 9, 11-я работы. Поэтому методика тестирования ПС включает следующую *последовательность шагов* [4, 32]:

1) ***Тестирование модулей*** (7-я работа).

2) ***Тестирование сборки*** (интеграции) (8-я работа).

3) ***Тестирование правильности*** (9-я работа).

4) ***Системное тестирование*** (11-я работа).

Процесс тестирования как правило начинается с тестирования отдельных модулей, а затем продолжается при объединении модулей в единое целое и завершается тестированием, при котором проверяется соответствие разработанного ПС требованиям заказчика.

Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рис. 8.14) [32].

На первом шаге осуществляется тестирование модулей, проверяющее результаты этапа кодирования. Целью данного шага является проверка каждого отдельного модуля. Используется как структурное, так и функциональное тестирование.

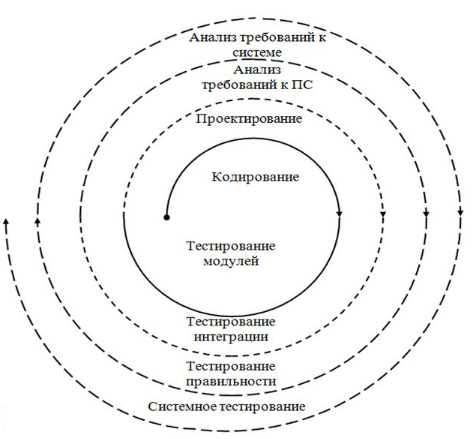


Рис. 8.14. Методика тестирования ПС

На втором шаге выполняется тестирование интеграции, направленное на выявление ошибок этапа проектирования ПС. Целью второго шага является сборка модулей в единое ПС и его тестирование. В основном используется функциональное тестирование и частично структурное тестирование.

На третьем шаге производится тестирование правильности, проверяется правильность этапа анализа требований к ПС. Целью третьего шага является проверка правильности реализации в ПС всех функциональных требований, а также требований по качеству к ПС. Проводится только функциональное тестирование.

На четвёртом шаге проводится системное тестирование, проверяется правильность этапа анализа требований к системе. Целью четвёртого шага является проверка реализации в ПС всех системных требований, а также требований по качеству к системе. Проводятся различные типы системного тестирования.

### **Тестирование модулей**

Проводится во время работы № 7 «Программирование и тестирование ПС» процесса разработки (см. подраздел 4.1). На данном шаге проверяется правильность функционирования наименьшего отдельно взятого элемента ПС. Что взято в качестве элемента (модуля) определяется контекстом проекта. Это может быть либо отдельно взятая функция, либо набор функций, либо отдельно взятый класс, либо набор классов и т.п. [4]

Вначале, с целью проектирования тестов, рекомендуется рассматривать модуль ПС как “чёрный ящик”, а затем для подготовки дополнительных тестов исследовать модуль как “белый ящик” [28, 29, 32].

Методика тестирования модулей включает следующие *шаги*:

1) руководствуясь спецификацией требований модуля, необходимо подготовить тесты для каждого входного условия (используются методы эквивалентного разбиения и граничных условий);

2) проверить текст модуля, чтобы убедиться, что все условные переходы будут выполнены в каждом направлении, т.е. каждая ветвь модуля будет выполнена. При необходимости спроектировать дополнительные тесты;

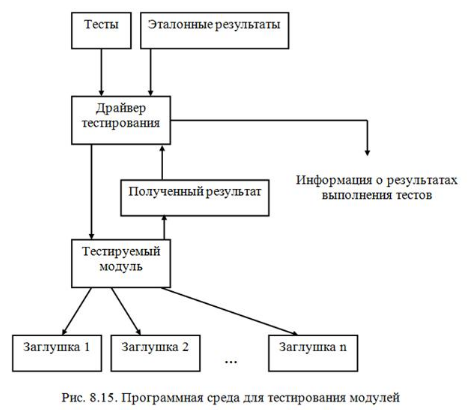
3) проверить по тексту модуля тот факт, что тесты охватывают достаточное количество путей (для каждого цикла должны быть составлены пути без выполнения тела цикла, с однократным и максимальным числом повторений;

4) проверить по тексту модуля его чувствительность к отдельным особым значениям входных данных и при необходимости спроектировать дополнительные тесты;

5) следует также спроектировать тесты, пытаясь выйти за границы упорядоченных структур данных (массивов, списков, файлов и т.д.).

Так как модуль является частью всего ПС, то для проведения его тестирования необходимо дополнительное ПС - драйвер и заглушка).

Программная среда для тестирования модулей имеет следующий вид (рис. 8.15).



*Драйвер* представляет собой управляющую программу, которая:

1) принимает тесты и ожидаемые результаты тестов (эталонные результаты);

2) осуществляет запуск в работу тестируемого модуля;

3) получает из модуля реальные результаты выполнения модуля;

4) формирует информацию о результатах тестирования.

*Заглушки* представляют собой фиктивные программы, замещающие модули, которые вызываются тестируемым модулем. Заглушка реализует интерфейс подчиненного модуля и может осуществлять минимальную обработку данных и имитировать прием и возврат данных.

В общем случае не все модули могут быть адекватно протестированы с помощью драйвера и заглушки. В таких случаях окончательное тестирование модуля может быть отложено до шага тестирования интеграции (сборки).

### **Тестирование сборки (интеграции).**

Тестирование сборки проверяет правильность взаимодействия между программными компонентами или модулями. Проводится во время работы № 8 «Сборка ПС» процесса разработки. В данной работе разрабатывается план сборки для объединения модулей и компонентов в единый программный объект [4].

План включает требования к тестированию, а также процедуры и методы тестирования. План должен быть документально оформлен. Затем, в соответствии с планом, модули и компоненты собираются в единое целое и тестируются. Выбор варианта тестирования сборки определяется той стратегией сборки, которая была выбрана при проектировании ПС.

Существуют различные *стратегии сборки*. Ограничимся рассмотрением двух основных [28, 29, 32]:

1) сборка модулей и компонент по мере движения по управляющей иерархии “снизу-вверх” (восходящее тестирование сборки, bottom-up);

2) сборка модулей и компонент по мере движения по управляющей иерархии “сверху-вниз” (нисходящее тестирование сборки, top-down).

При выполнении данной работы в основном находятся ошибки интерфейса, например:

– потеря данных при прохождении через интерфейс;

– «неблагоприятное» влияние одного модуля на другой;

– проблемы при работе с глобальными переменными.

### **Тестирование правильности. Системное тестирование. Типы системного тестирования.**

### **Тестирование правильности**

На данном этапе (работа № 9 процесса разработки по стандарту СТБ *ИСО/МЭК 12207-2003*, см. подраздел 4.1) проводится тестирование на соответствие требований ко всему программному объекту. Данные требования прописаны в спецификации требований к ПС (работа № 4, см. подраздел 4.1) [4]. Используется только функциональное тестирование, которое должно подтвердить, что функции, прописанные спецификацией, работают правильно.

В общем, разработчик не может знать, как заказчик будет реально использовать разработанное ПС. Для обнаружения дополнительных ошибок, которые может найти заказчик или конечный пользователь, используется Альфа- и Бета-тестирование [28, 29, 32].

Альфа-тестирование проводится заказчиком или конечным пользователем в организации разработчика с документацией всех выявленных ошибок. Это могут быть сотрудники организации разработчика, но не участвовавшие в разработке ПС.

Бета-тестирование проводится конечным пользователем, а так же заказчиком. Заказчик проводиттестирование в своей организации, при этом фиксируются и документируются все ошибки. Затем по результатам всех выявленных ошибок разработчик корректирует ПС, повторно его тестирует и (при положительных результатах тестирования) отдает его в эксплуатацию заказчику или конечному пользователю

### **Системное тестирование**

В работе № 10 процесса разработки (Сборка системы, см. подраздел 4.1) объекты программной конфигурации собираются в единую систему вместе с объектами технической конфигурации и при необходимости с другими системами. Система тестируется, начиная с работы № 11 (см. подраздел 4.18.11), и проводится в соответствии с требованиями, установленными к системе. При этом должна быть обеспечена проверка каждого требования к системе на соответствие заданным требованиям [4]. Только при системном тестировании можно оценить характеристики ПС, которые нельзя оценить на более ранних этапах тестирования.

К основным ***типам системного тестирования*** относятся [32]:

1. *Тестирование производительности*. Существует огромное количество систем реального времени, в которых должны жестко выполняться требования по производительности. Например, системы, управляющие технологическими процессами, обучающие программы, игры и т.п. Только системное тестирование проверяет истинную производительность всей системы в целом.

2. *Тестирование надежности*. Проверяется правильность работы всей системы при заданных условиях для заданного интервала времени. Надёжность можно измерить метриками:

- средняя наработка на отказ;

- вероятность безотказной работы и т.п.

3. *Тестирование восстановления*. Проверяется отказоустойчивость системы. Система должна быть быстро восстанавливаема после сбоев и отказов. При этом отказы не должны быть причиной прекращения работы всей системы, т. к. отказ для отдельных систем может привести к огромному ущербу.

4. *Стрессовое тестирование*. Стрессовые тесты проектируются таким образом, чтобы проверить работу всей системы в ненормальных ситуациях. Например, при ненормальных запросах на ресурсы системы:

- по количеству запросов;

- по частоте запросов;

- по размеру используемых ресурсов и т.п.

Целью стрессового тестирования является нахождение ошибок, появление которых спровоцировано дефицитом ресурсов, т.е. нехваткой свободной оперативной памяти или свободного места на внешнем носителе, или нехваткой пропускной способности сети.

5. *Тестирование безопасности*. При этом можно подвергнуть систему попыткам НСД (несанкционированного доступа) и при этом измерять среднее время взлома. Взлом лучше всего организовать с помощью внешних средств.

6. *Тестирование практичности*. Оценивается “дружелюбие'' разработанного ПС к пользователю и фиксируются те операции, которые могут вызвать трудности у пользователя.

Тесты практичности включают тесты на:

а) наличие и качество оперативной и контекстной помощи;

б) наличие руководств и учебных материалов;

в) проверку на соответствие общепринятым стандартам графического интерфейса и т.п.

### **Регрессионное тестирование.**

Основной работой тестировщиков является ***регрессионное тестирование***. Оно основано на повторном использовании ранее разработанных тестов. Тестировщик после обнаружения ошибки передает отчет о ней программисту. В отчете приводятся симптомы проявления ошибки. Программист анализирует отчет, исправляет данную ошибку и отдает исправленное ПС назад тестировщику. Тестировщик затем опять проверяет ПС на том же самом тесте, который ранее приводил к ошибке, так как должен убедиться в том, что ошибка исправлена и больше не проявляется [25].

Реально программист может исправить только описанные в отчете симптомы, и фактически могут исправляться только отдельные причины проявления ошибки. Кроме того, программист может не правильно понять отчет об ошибке и исправить не то, что надо. Таким образом, после исправления одни ошибки могут устраняться, а другие оставаться.

Статистика показывает, что в среднем каждое третье исправление ошибки приводит к внесению новых ошибок. Поэтому очевидно, что после исправления программистом каждой ошибки, тестировщик должен спроектировать дополнительные тесты. Целью данных тестов является проверка того, чтобы убедиться, что при исправлении ошибки не внесены новые ошибки. Для этого необходимо проанализировать те части ПС, которые могут быть затронуты внесением изменений.

Кроме того, рекомендуется проверить целостность всего ПС, то есть правильность его работы, на следующих тестах:

1) перечисленные тесты в начальном плане тестирования;

2) тесты, которые выявляли все предыдущие обнаруженные ошибки;

3) тесты, сгенерированные случайным образом.

Таким образом, повторное выполнение тестов (регрессионное тестирование) необходимо для проверки того, что изменения, внесенные в программу в результате устранения ошибок или изменения существующей функциональности, не повлияли на правильность функционирования тестируемой программы.

### **Документирование ошибок. Структура отчета об ошибке.**

В общем программная ошибка(или баг) представляет собой некоторый дефект в разработке программного продукта, который вызывает несоответствие ожидаемых результатов выполнения программного продукта и фактически полученных результатов.

В общем задокументировать ошибку может практически любой разработчик или пользователь, обнаруживший некорректное поведение программы. Но собственно написание отчёта об ошибке («bugreport») является одним из основных результатов работы тестировщика. Основная цель написания отчёта об ошибке - устранение ошибки. В общем отчёт об ошибке представляет собой технический документ, написанный с целью:

- предоставить информацию о возникшей проблеме, её свойствах и последствиях;

- назначить приоритет возникшей проблеме с учётом её важности и скорости устранения;

- помочь программистам обнаружить и устранить источник проблемы.

Для формирования отчёта об ошибке используются так называемые багтрэкинговые системы.

Часто данные системы интегрируются с системами управления проектами, так как ошибка в разрабатываемом приложении является угрозой для качественного выполнения проекта в целом. Типовая стандартная форма отчёта об ошибках, используемая в ведущих IT-компаниях, содержит следующие основные поля, которые заполняются как тестировщиком, так и менеджером проекта [25]:

1) Краткое описание(Summary). Содержит краткое описание ошибки, возникшей в приложении. Чаще всего указывается место, где произошла ошибка, и вкратце её суть.

2) Версия приложения, в котором была обнаружена ошибка(Build found) .

3) Симптом (Symptom) ошибки. Это поле показывает вид ошибки, т.е. как она проявляется. Наиболее широко распространённые симптомы:

- некорректная операция (incorrect operation). Например, неверная арифметическая операция;

- повреждение/потеря данных (data corruption/loss). В результате ошибки данные повреждаются или теряются.

- крах системы (system crash). Например, приложение или операционная система виснет, перезагружается или закрывается.

- косметический дефект (cosmetic flaw). Например, опечатки, повреждённые картинки, не тот цвет и т.п.

- проблема в документации (documentation issue). Если ошибка описывает проблему не в приложении, а в документации.

- нереализованная функциональность (missing feature).

4) Серьёзность (Severity) ошибки. Это поле показывает, насколько серьёзна найденная ошибка. Обычно выделяют следующие уровни серьёзности:

- критическая (critical) ошибка. Это самые серьёзные ошибки, ведущие к краху приложения или операционной системы, серьёзному повреждению базы данных, падению веб-сервера или сервера приложений;

- высокая (major) ошибка. Это также серьёзные ошибки, ведущие к потере/искажению данных, падению значительной части функциональности приложения, падению браузера и т.п.;

- средняя (medium) ошибка. Это ошибки, которые не приводят к краху системы или потере данных. Они обычно затрагивают небольшой набор функциональности, чаще всего это некорректная обработка вводимых значений. Как правило, такие ошибки можно «обойти», т.е. выполнить требуемое действие иным способом, не приводящим к возникновению ошибки;

-низкая (minor) ошибка. Как правило, это ошибки «косметического» плана. Они непосредственно не мешают работе с приложением.

5) Приоритет (Priority) ошибки. Это поле показывает, как быстро необходимо исправить ошибку. Обычно выделяют следующие уровни приоритета:

- наивысший (ASAP, as soon as possible). При этом ошибка должна быть исправлена немедленно, так как её наличие делает невозможным дальнейшую работу над проектом;

- высокий (high). Обычно присваивается ошибкам, которые нужно исправить как можно быстрее;

- обычный (normal). Ошибка должна быть исправлена в ближайшем будущем после исправления всех ошибок с более высоким приоритетом;

- низкий (low). Это самые низкоприоритетные ошибки, которые исправляются в последнюю очередь.

Если серьёзность ошибки выставляет тестировщик, то приоритет ошибки выставляет менеджер проекта после просмотра отчёт об ошибке.

6) Воспроизводимость (Reprodusible) ошибки. Это поле показывает, как воспроизводится ошибка. Возможны два уровня воспроизводимости ошибки:

-всегда (always), когда ошибка воспроизводится постоянно;

-иногда (sometimes), когда ошибка воспроизводится при определённых условиях.

7) Шаги для воспроизведения ошибки (Steps to reproduce). Данное поле приводится для того, чтобы воспроизвести ошибку в кратчайшие сроки. Это поле следует заполнять максимально подробно.

8) Подробное описание ошибки (Description). Приводится для того, чтобы программист чётко понял суть проблемы. Чем подробнее описание, тем быстрее причина ошибки будет обнаружена и тем быстрее ошибка будет исправлена.

Кроме словесного описания в поле Discription тестировщик может «прикрепить» Screen Shot, изображающий полученную ошибку. Хорошее описание содержит необходимую информацию об ошибке, описание ожидаемого результата, описание полученного результата и ссылку на требование.

### **Жизненный цикл дефекта.**

Рассмотрим рис. 8.19, иллюстрирующий жизненный цикл дефекта (бага, ошибки), принятый во многих крупных IT компаниях.

Баг может находится в одном из представленных на рисунке состояний.

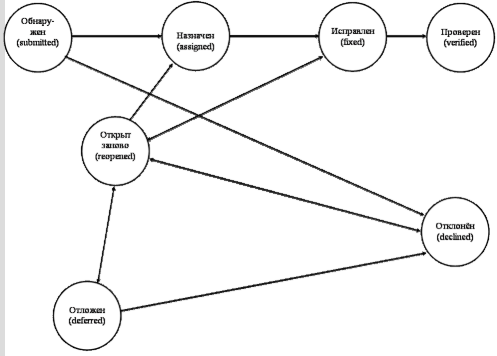


Рис. 8.19. Жизненный цикл ошибки

После обнаружения тестировщиком дефекта он вносится в баг-трэкинговую систему и ему присваивается статус **Обнаружен (submitted)**.

Итак, тестировщик находит дефект и представляет его на рассмотрение в систему управления дефектами. С этого момента баг начинает свою официальную жизнь и о его существовании знают необходимые люди.

Далее ведущий разработчик, например, менеджер, рассматривает дефект, присваивает ему приоритет и назначает его исправление кому-то из команды разработчиков (программистов) – дефекту присваивается статус **Назначен (assigned).**

Разработчик, которому было назначено исправление дефекта, исправляет его и сообщает о том, что задание выполнено. Дефекту присваивается статус **Исправлен (fixed).**

Тестировщик, который обнаружил ошибку, проверяет на новой версии продукта (в которой исправление данной ошибки заявлено), исправлен ли дефект на самом деле. И только в том случае, если ошибка не проявится в новой версии, тестировщик меняет статус бага на **Проверен (verified)** и закрывает дефект.

Если баг проявляется в новой версии, тестировщик снова открывает этот дефект. Баг приобретает статус **Открыт заново (reopened).**

Баг может быть отклонён. Во-первых, потому, что для заказчиков какие-то ошибки перестают быть актуальными. Во-вторых, это может случиться по вине тестировщика из-за плохого знания продукта, требований (дефекта на самом деле нет). Дефекту присваивается статус **Отклонён (declined).**

Если исправление конкретного бага сейчас не очень важно или заказчик пока думает, или мы ждём какую-то информацию, от которой зависит исправление бага, тогда баг приобретает статус **Отложен (deferred).**

**Закрытым** считается баг в состоянии **Проверен (verified)** и **Отклонён (declined)**.

**Открытыми** являются баги в состоянии **Обнаружен (submitted), Назначен (assigned), Открыт заново (reopened)**. Иногда к открытым относят и баги в состояниях **Исправлен (fixed)** и **Отложен (deferred).**

### **Проектирование тестов, основанных на случайных наборах исходных данных. Получение закона распределения по заданным моментам исходного распределения.**

В процессе разработки интерес представляет оценка надежности промежуточного продукта. На ранних этапах разработки это может быть спецификация требований, архитектура или технический проект программного продукта, исходные коды модулей. Для них выполняется оценка надежности по внутренней модели надежности с целью прогноза внешнего уровня надежности.

На последующих этапах процесса разработки промежуточными продуктами являются исполнимые коды модулей и промежуточных продуктов сборки, а также конечный программный продукт. Для них выполняется оценка надежности по внешней модели в моделируемой среде с моделируемыми исходными данными с целью прогноза уровня надежности программного продукта в среде эксплуатации. При этом моделируемые данные должны быть адекватными реальным данным в заданных условиях эксплуатации.

Метод моделирования случайных наборов данных основан на получении закона распределения в виде функции плотности.

### **Получение закона распределения по заданным моментам исходного распределения**

Методы генерации значений случайных величин используют заданный закон распределения в виде известной функции распределения или функции плотности. Но часто аналитический вид функции распределения или функции плотности неизвестен, а известен ряд моментов исходного распределения (или некоторая ограниченная *выборка*). В этом случае плотность или функцию исходного распределения можно представить в виде разложения по полиномам.

Наиболее общим способом представления скалярных и векторных случайных величин является представление их в форме линейных комбинаций некоторых эталонных функций.

Разложение искомой функции плотности обычно ищут в виде ряда



где: ***Сn*** - некоторый коэффициент разложения;

- весовая (эталонная) функция плотности;

***Qn(х)***- система ортогональных полиномов.

Выбор определяется характером исходного распределения. Если распределение исходных данных на заданном интервале близко к нормальному, то хорошие результаты дает разложение, основанное на нормальном законе. В качестве обычно берется нормальная функция плотности нормированной случайной величины (- для удобства расчета)



В качестве ***Qn(x)***используется ***полином Эрмита n-порядка***, который определяется как

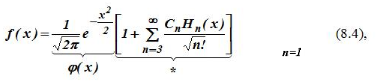


где:

***n*** = 0, 1, 2, ...

производная ***n***-порядка

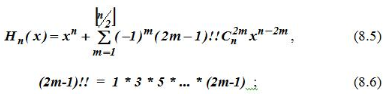
Выражение аппроксимирующей функции (8.1) имеет вид



где:

\* - ряд, учитывающий асимметрию распределения;

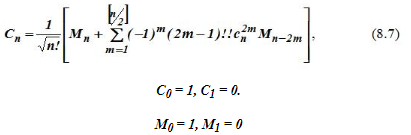
***Hn(х)***- полином Эрмита ***n***-порядка, определяемый как



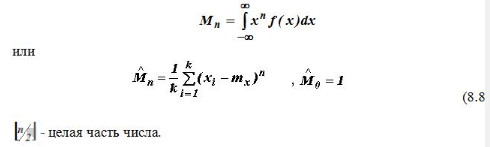
в выражении (8.5) число сочетаний из ***n*** по ***2m*** определяется формулой:



***Cn*** - *коэффициент разложения*, определяемый как



В выражении (8.7) ***Mn*** *- центральный момент* ***n****-порядка:*

**

Обычно ряд (8.4) называется *рядом Грама-Шарлье типа А*. В том случае, когда моменты исходного распределения не заданы, а известна выборка значений исходных распределений, в качестве центральных моментов берутся их оценки (8.8).

Из анализа выражения (8.7) для вычисления ***Cn*** видно, что необходимо знание центральных моментов исходного распределения до ***n***-го порядка включительно.

Вопрос о сходимости ряда (8.4) не имеет значения; важно быть уверенным, что первые слагаемые данного ряда дают достаточно точное приближение.

Как показывает анализ литературы, аппроксимация исходного распределения с помощью *ряда Грама-Шарлье типа А* тем лучше, чем ближе исходное распределение к нормальному.

### **Проектирование тестов, основанных на случайных наборах исходных данных. Генерация случайных значений исходных данных.**

### **Генерация случайных значений исходных данных**

Процедура генерации случайных значений в соответствии с заданной функцией плотности ***f(x)*** заключается в следующем [35].

Функция плотности ***f(x)*** аппроксимируется ступенчатой функцией и при этом отрезок **[*a, b*]**разбивается на ***n*** отрезков (рис. 8.20).

1) Генерируется равномерно распределенное случайное число



и определяется число**,** равномерно распределенное на отрезке [a, b]:

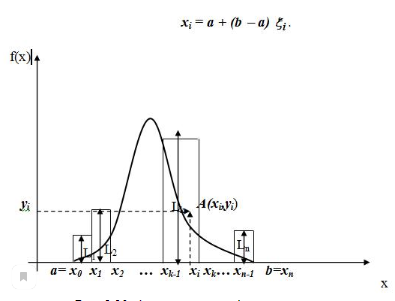


Рис. 8.20. Аппроксимация функции плотности

2) Значение индекса ***k*** соответствует попаданию случайного числа ***xi*** в некоторый интервал



значение ***Lk*** равно высоте выбранной ступеньки.

3) Генерируется следующее случайное число, которое принадлежит отрезку [0, 1],



и определяется случайное число



4) Если точка ***А*** с координатами ***А(xi, yi)*** находится под кривой ***f(x)***, то ***xi*** является реализацией случайной величины ***X***, имеющей заданное распределение ***f(x).*** А если точка ***А*** с координатами ***А(xi, yi)***находится над кривой ***f(x)***, то ***xi*** не является реализацией случайной величины ***X***, имеющей заданное распределение ***f(x).***

5) Шаги 1, 2, 3, 4 повторяются до тех пор, пока не получится требуемое число реализаций случайной величины ***X***, имеющей заданное распределение ***f(x).***

### **Понятие верификации. Понятие высказывания, высказывательной формы и предиката. Операция импликации. Примеры.**

Согласно стандарту *СТБ ИСО 12207-2003* процесс верификации является процессом того, что программные продукты функционируют в полном соответствии с требованиями или условиями, реализованными в предшествующих работах [4].

Под ***верификацией*** понимают подтверждение экспертизой и представлением объективных доказательств того, что конкретные требования полностью реализованы

В общем, под ***верификацией*** понимается процесс проверки правильности какой-то работы ЖЦ. Чем раньше начать верификацию работ, тем выше будет качество будущего ПО, тем раньше будут найдены все дефекты при разработке ПО.

В данном разделе ограничимся рассмотрением верификации применительно к доказательству правильности алгоритмов и программ.

Верификация программ состоит в формальном доказательстве их правильности. В отличие от тестирования, имеющего дело со свойствами отдельных процессов программы, верификация имеет дело со свойствами всей программы и для всей совокупности исходных данных.

**Элементы математической логики**

**Исчисление высказываний**

***Исчисление высказываний*** - это совокупность правил для определения истинности или ложности высказываний.

***Высказывание*** - это предложение, которое либо истинно, либо ложно.

Предложение ***x2 = 4***, например, не является высказыванием, т.к. чтобы говорить об истинности или ложности высказывания, нужны дополнительные сведения (чему равно число ***x***). В этом предложении ***x*** - некоторая переменная, вместо которой можно подставить элементы некоторого множества, называемые значениями этой переменной.

Предложение, которое содержит хотя бы одну переменную и становится высказыванием при подстановке вместо всех переменных их значений, называется ***высказывательной формой*** (**ВФ**).

В математической логике всякое предложение, составленное из высказываний с помощью логических связок (и; или; если то; тогда и только тогда, когда и т.п.), становится высказыванием.

**Операции над высказываниями**

1. Конъюнкция

***A Ù B (A&B)***

2. Дизъюнкция

***A ÚB (A+B)***

1. Отрицание

***~A* ()**

2. Импликация

***(если A, то B)***

Определение импликации иллюстрирует таблица 9.1.

Таблица 9.1.

Определение импликации

Определение импликации соответствует употреблению союза "*если-то*" не только в математике, но и в обыденной речи. По определению логической операции смысл составляющих высказываний не учитывается. Составляющие высказывания рассматриваются как объекты, обладающие единственным свойством: *быть истинными или ложными*.

**Исчисление предикатов**

Исчисление предикатов является расширением логики высказываний.

Рассмотрим высказывательную форму ***sin(x)=1.*** Данная форма каждому ***x*** на множестве действительных чисел ставит в соответствие некоторое высказывание и тем самым одно из значений истинности. Таким образом, данная высказывательная форма задаёт отображение множества действительных чисел ***R*** на множество значений {***Истина, Ложь***}, то есть задаёт функцию с областью определения ***R*** и множеством значений ***{Истина, Ложь}***.

Функция, все значения которой принадлежат множеству ***{Истина, Ложь},*** называется ***предикатом***. Чаще всего предикаты задаются с помощью высказывательных форм.

Например,

- одноместная высказывательная форма,

- предикат с двумя неизвестными,

- трехместная высказывательная форма.

**Операции над высказывательными формами**

**Конъюнкцией** высказывательных форм ***Ф1*** и ***Ф2*** называется высказывательная форма, истинная при тех и только при тех значениях входящих в неё переменных, которые обращают обе формы в истинное высказывание.

**Дизъюнкцией** высказывательных форм ***Ф1*** и ***Ф2*** называется высказывательная форма, ложная при тех и только при тех значениях входящих в неё переменных, которые обращают обе формы в ложное высказывание.

**Отрицанием** высказывательной формы ***Ф*** называется высказывательная форма, ложная при тех наборах значений переменных, которые обращают высказывательную форму в истинное высказывание и наоборот.

**Импликацией** высказывательных форм ***Ф1*** и ***Ф2*** называется высказывательная форма ***Ф1Ф2***, ложная при тех и только при тех значениях входящих в неё переменных, которые обращают ***Ф1*** в истинное высказывание, а ***Ф2*** - в ложное.

### **Метод индуктивных утверждений. Общие сведения.**

***Верификация*** ПО заключается в выполнении формального доказательства того, что ПО удовлетворяет своей спецификации [14].

Общими для различных методов ***верификации*** ПО является представление программ в виде схемы алгоритма, с каждой дугой которой соотносится некоторый предикат, называемый утверждением (рис. 9.1).

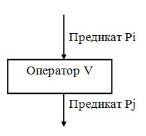


Рис. 9.1. Пример фрагмента схемы алгоритма с предикатами

Если ***Pi***есть утверждение, связанное с входящей дугой оператора ***V,*** а ***Pj*** – утверждение, связанное с исходящей дугой того же оператора, то тогда необходимо доказать правильность следующего оператора: если ***Pi*** истинно и если оператор ***V*** выполнен, то утверждение ***Pj*** истинно. Такие теоремы имеют следующий вид: ***Pi→Pj***.

Обычный метод их доказательства сводится, например, к тому, чтобы показать, что***Pj*** истинно всякий раз, когда ***Pi***истинно, т.е. для доказательства всей теоремы необходимо доказать, например, истинность ***Pj***. Подобный процесс может быть повторен для каждого оператора программы.

На рис. 9.2 приведена схема алгоритма некоторой программы с предикатами ***P1*** и ***Pn***.

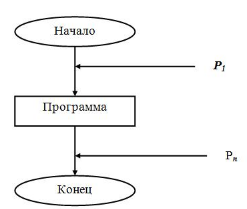


Рис. 9.2. Пример схемы алгоритма некоторой программы с предикатами

На данном рисунке ***P1*** – утверждение, непосредственно предшествующее входному оператору программы (начальное утверждение), а ***Pn*** – утверждение, соответствующее выходному оператору программы (конечное утверждение).

Тогда оператор «если ***P1*** истинно и программа выполнена, то ***Pn*** истинно», представляет собой теорему, доказательство которой устанавливает правильность того, что ПО соответствует своим спецификациям (т.е. теорема имеет вид ***P1→Pn***). Таким образом, доказательство правильности программ сводится к доказательству теорем методами исчисления предикатов.

Одним из известных методов доказательства правильности программ является *метод индуктивных утверждений*, предложенный Флойдом.

Данный метод применим для доказательства правильности или корректности схем алгоритмов. В понятии корректности программ можно выделить два аспекта:

1). Частичная корректность.

2). Завершаемость.

***Частичная корректность*** подразумевает соответствие программы своей спецификации.

***Завершаемость*** означает, что выполнение программы закончится при вводе определенных данных в соответствии со спецификацией. Нет общего метода доказательства завершаемости программ, но обычно бывает достаточно неформальных рассуждений.

В методе индуктивных утверждений первый шаг состоит в записи утверждений относительно свойств входных и выходных данных программ, а так же результатов в ряде промежуточных точек называемых точками разреза. Эти утверждения формулируются в логике исчисления предикатов. На основании этих утверждений и семантики операторной схемы программы путем определенных преобразований формулируются верификационные условия, правильность которых необходимо доказать. И если доказывается истинность сформулированных условий, то программа будет корректна относительно входного и выходного утверждений. Если доказать истинность не удается, то либо в ПС есть ошибка, либо в процедуре доказательства есть ошибка, например, неверное утверждение в некоторой точке разреза.

### **Процесс верификации в жизненном цикле ПО.**

Стандарт СТБ *ИСО/МЭК 12207-2003* дает такое определение процесса верификации [4].

***Процесс верификации*** ***(The Verification Process)*** является процессом определения того, что программные продукты функционируют в полном соответствии с требованиями и условиями, реализованными в предшествующих работах.

Верификация может применяться к любым промежуточным программным продуктам и к любым другим результатам работы. Например, верификации могут подвергаться требования, системная и программная архитектура, документация, методы, планы и т.п.

Процесс верификации может включать анализ, проверку и тестирование. Объектами анализа и проверки могут являться, например, документация и исходные тексты программных модулей. Объектами тестирования могут являться исполнимые коды программных модулей, компонентов, промежуточных и конечного программных продуктов.

Данный процесс называется ***процессом независимой верификации***, если организация–исполнитель не зависит от поставщика, разработчика, оператора или персонала сопровождения.

Процесс верификации по СТБ *ИСО/МЭК 12207-2003* состоит из *двух работ* (рис. 9.5). Общее число задач по данным работам равно 13.

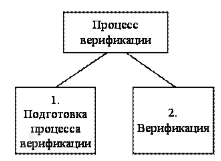


Рис. 9.5. Структура процесса верификации

Табл. 9.2 содержит задачи, реализуемые при выполнении соответствующих работ процесса верификации, и типы выходных результатов данных задач, которые должны быть документально оформлены.

Таблица 9.2

Задачи работ процесса верификации

|  |  |  |
| --- | --- | --- |
| № задачи | Описание задачи | Тип выходного результата задачи |
| 1. Задачи работы ***Подготовка процесса верификации*** | | |
| 1.1 | Определение необходимости в проекте работ по верификации и степени их организационной независимости, анализ критичности проектных требований | Описание  (в составе  результата  задачи 1.5) |
| 1.2 | Установка процесса верификации (при необходимости) | Описание (в составе результата 1.5) |
| 1.3 | Выбор соответствующей независимой квалифицированной организации (при необходимости) | Описание (в составе результата 1.5) |
| 1.4 | Определение верифицируемых работ и продуктов, выбор работ и задач верификации | Описание (в составе результата 1.5) |
| 1.5 | Разработка плана верификации | План |
| 1.6 | Реализация плана проведения верификации, обеспечение доступности его результатов заказчику, при необходимости организация связи с процессом решения проблем (см. п. 2.3.8 пособия) | Отчет,  протокол |
| 2. Задачи работы ***Верификация*** | | |
| 2.1 | Верификация договора по критериям |  |
| 2.2 | Верификация процесса по критериям |  |
| 2.3 | Верификация требований по критериям |  |
| 2.4 | Верификация результатов проектирования по критериям |  |
| 2.5 | Верификация исходных текстов программных модулей по критериям |  |
| 2.6 | Верификация сборки по критериям |  |
| 2.7 | Верификация документации по критериям |  |

***Договор*** должен быть верифицирован (см. задачу 2.1) по следующим *критериям*:

· возможность поставщика удовлетворять установленным требованиям;

· непротиворечивость требований и охват ими потребностей пользователя;

· наличие соответствующих процедур для внесения изменений в установленные требования и для решения проблем;

· наличие процедур по взаимодействию и кооперации между участниками договора;

· наличие критериев и процедур, предусмотренных в соответствии с установленными требованиями.

***Процесс*** должен быть верифицирован (см. задачу 2.2) по следующим *критериям*:

· соответствие и своевременность установления требований к планированию проекта;

· пригодность, реализуемость, выполнимость в соответствии с планом и условиями договора выбранных для проекта процессов;

· применимость стандартов, процедур и условий к процессам проекта;

· укомплектованность и обученность персонала в соответствии с условиями договора.

***Требования*** должны быть верифицированы (см. задачу 2.3) по следующим *критериям*:

· непротиворечивость, выполнимость и тестируемость требований к системе;

· распределение требований к системе между объектами технических и программных средств и ручных операций в соответствии с критериями проектирования;

· непротиворечивость, выполнимость, тестируемость и точность отражения требований к системе в требованиях к программным средствам;

· правильность (подтвержденная соответствующими методами) критических требований к программным средствам, в том числе по безопасности и защите.

***Результаты проектирования*** должны быть верифицированы (см. задачу 2.4) по следующим *критериям*:

· правильность, соответствие установленным требованиям и учет этих требований;

· реализация соответствующей последовательности событий, исходных данных, выходных результатов, интерфейсов, логики; соответствие временным ограничениям и ограничениям размера; обнаружение, локализация и устранение ошибок;

· возможность дальнейшего использования с учетом требований;

· правильность, подтвержденная соответствующими методами, реализации требований безопасности, защиты и других критических требований.

***Исходные тексты программных модулей*** должны быть верифицированы (см. задачу 2.5) по следующим *критериям*:

· трассируемость с результатами проектирования и требованиями; тестируемость, правильность и соответствие требованиям и стандартам программирования;

· реализация соответствующей последовательности событий, соответствующих интерфейсов, правильных данных и логики управления; завершенность; соответствие временным ограничениям и ограничениям размера; обнаружение, локализация и устранение ошибок;

· соответствие результатам проектирования и требованиям;

· возможность дальнейшего использования с учетом результатов тестирования и требований;

· правильность, подтвержденная соответствующими методами, реализации требований безопасности, защиты и других критических требований.

***Сборка*** должна быть верифицирована (см. задачу 2.6) по следующим *критериям*:

· полнота и правильность сборки программных компонентов и модулей каждого программного объекта в соответствующий программный объект;

· полнота и правильность сборки технических и программных объектов и ручных операций в систему;

· выполнение задач сборки в соответствии с планом сборки.

***Документация*** должна быть верифицирована (см. задачу 2.7) по следующим *критериям*:

· соответствие, полнота и непротиворечивость документации;

· своевременность подготовки документации;

· соблюдение установленных процедур управления конфигурацией документов.

### **N-версионное программирование.**

Очевидно, что использование технологий, минимизирующих ошибки в ПО, методов верификации, методов тестирования и отладки, позволяет существенно уменьшить количество ошибок ПО, но к сожалению все программные ошибки не устраняются. Оставшиеся скрытые ошибки ПО могут при определенном сочетании исходных данных привести к отказам в ходе эксплуатации.

Естественным является стремление повысить отказоустойчивость ПО, применяемого в критических областях путём использования различных форм избыточности. Введение форм избыточности является классическим приемом обеспечения, например, надежной работы аппаратуры. Специфика программной избыточности определяется разной природой неисправности. Если в аппаратуре сбой и отказы имеют очевидно случайный характер вследствие, например:

- локального перегрева;

- окисления контактов;

- пробой **p-n** перехода;

- старение полупроводника;

- электромагнитные помехи и т. д.,

то наработка на отказ для аппаратуры представляет собой случайную величину (СВ), имеющую некоторый закон. В ПО ошибки определяются в первую очередь его сложностью и вносятся как будто неслучайно, практически во всех работах жизненного цикла ПО. Но проявляются ошибки в процессе эксплуатации в случайный момент времени при случайном сочетании исходных данных, и таким образом, можно считать, что наработка на отказ так же представляет собой СВ. Но природа ошибки в ПО совершенно другая.

Под ***N–версионным программированием*** понимается независимое создание ***N ≥ 2*** функционально–эквивалентных программ на основе общего исходного описания.

Можно сформулировать следующие ***требования к программной избыточности*** (т.е. к ***N***–версионномупрограммированию):

1. Для выполнения одной и той же задачи рекомендуется разработать ***N*** независимых программ. При этом независимое создание программ подразумевает отсутствие контактов в процессе работы между ***N*** программистами или группами программистов. При этом насколько возможно, желательно использование различных алгоритмов, языков программирования и средств разработки.

2. В спецификации ПО необходимо так же предусмотреть дополнительно:

а) разработку алгоритма сравнения по совпадению при ***N = 2*** и мажоритарные при ***N > 2***;

б) точки контроля;

в) для оценки правильности результатов в алгоритме сравнения устанавливается допустимый диапазон расхождения числовых результатов.

3. ***N*** независимо разработанных программ должны работать параллельно на ***N*** независимых компьютерах. Затем результаты вычисления на каждом компьютере сравниваются между собой, и в случае расхождения наиболее вероятный результат находится голосованием (по большинству) для ***N > 2***. При этом на аппаратном уровне:

а) компьютеры должны иметь эффективные средства взаимодействия с целью быстрого сравнения результатов;

б) архитектура системы должна быть такова, чтобы отказ одного из компьютеров не мог вызвать отказа всей системы в целом.

Данная схема сравнения результатов вычислений соответствует *мажоритарному резервированию* или *резервированию по принципу голосования*. Возможны следующие схемы: 2 из 3, 3 из 5, 4 из 7 и т.д. Например, схема 2 из 3 имеет следующий вид, приведенный на рис. 10.1.

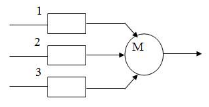


Рис. 9.5. Схема голосования “два из трех”

На данном рисунке: 1, 2, 3 – идентичные системы обработки информации (компьютер); М – мажоритарный орган.

Чем больше параллельно работающих элементов, тем выше уровень надежности и отказоустойчивости. Чем больше число элементов, результаты которых должны совпадать, тем выше достоверность. Для схемы 2 из 3 вместо одного элемента включается три идентичных элемента, выходы которых подаются на мажоритарный орган, который осуществляет операцию выбора по большинству. Условием безотказной работы данной системы является безотказная работа любых двух элементов из трех и мажоритарного органа. Вероятность безотказной работы данной системы равна:



при условии равнонадежности элементов 1, 2, 3. Здесь: ***Р*** – вероятность безотказной работы каждого из элементов 1, 2, 3; ***q*** – вероятность отказа каждого из элементов 1, 2, 3; ***РМ*** – вероятность безотказной работы мажоритарного органа.

Такая схема не позволяет достичь вероятности безотказной работы системы выше, чем вероятность мажоритарного органа **PМ**, но вероятность **PМ** в силу простоты мажоритарного органа намного выше, чем вероятность безотказной работы элементов 1, 2, 3. В общем сам мажоритарный орган можно зарезервировать. Например, идентичные программы в центральном вычислительном комплексе космического челнока Space Shuttle выполняются на трёх и более процессорных блоках с последующим мажоритарным выборам.

Очевидно, что N-версионное программирование позволяет бороться с ошибками как на программном, так и на аппаратном уровне. Объективный недостаток – стоимость разработки увеличивается в **N** раз.

### **Отказоустойчивость компьютерных систем. Граф процесса восстановления типичной отказоустойчивой системы**

Отказоустойчивость – свойство архитектуры компьютерных систем, позволяющее пользователю или программе продолжить работу и тогда, когда в аппаратных или программных средствах возникли отказы.

По способу реализации отказоустойчивость подразделяется на [23]:

1) активную;

2) пассивную.

Активная отказоустойчивость базируется на отдельно выделенных процессах обнаружения отказа, локализации отказа и реконфигурации системы.

Отказы обнаруживаются при помощи средств контроля, локализуются при помощи средств диагностирования и устраняются автоматически реконфигурацией системы, которая заключается в перестройке структуры компьютерных систем таким образом, чтобы её отказавшие части были устранены от участия в работе.

Пассивная отказоустойчивость заключается в свойстве системы не потерять свои функциональные свойства в случае отказа отдельных элементов системы. Иногда говорят, что отказ маскируется системой.

Пример пассивной отказоустойчивости систем - системы с мажоритарным органом. Пассивная отказоустойчивость связана с увеличением аппаратуры в несколько раз.

Пассивная отказоустойчивость применяется в случае особенно ответственных компьютерных систем, в случаях, когда не допустимы даже кратковременные перерывы в работе компьютерных систем, а также для обеспечения отказоустойчивости важнейших подсистем компьютерных систем.

Применение активной отказоустойчивости характеризуется более экономным расходом аппаратных средств, чем применение пассивной отказоустойчивости. Но оно связано с восстановлением работы системы после отказа, а также возможными потерями некоторой части данных.

Активная отказоустойчивость реализована только в многопроцессорных системах.

В то же время применение пассивной отказоустойчивости гарантирует практически безостановочную работу компьютерных систем и сохранение всей информации.

Введение отказоустойчивости является одним из методов повышения надежности компьютерных систем. Вопрос о построении и применении отказоустойчивых систем возникает тогда, когда другие пути повышения надежности не могут обеспечить требуемого уровня надежности или тогда, когда они оказываются экономически не оправдываемыми.

Отказ компьютерных систем может быть вызван различными причинами. Кроме отказов встречаются сбои, вызванные случайными помехами и разрушающие обрабатываемую информацию в течении одного или нескольких тактов работы. Отказы и сбои вызывают ошибки в работе элементов аппаратуры. Кроме ошибок аппаратуры встречаются ошибки оператора (входные данные) и ошибки программ, которые также могут исказить обрабатываемую информацию. Множество выше перечисленных ошибок может привести к невыполнению компьютерными системами своих функций. Отказом компьютерных систем также может быть понижение производительности системы ниже допустимого уровня, вследствие отказов определенных подсистем.

В общем отказоустойчивость системы обеспечивается введением избыточности, т.е. созданием определенных резервов или запасов в системе. В компьютерных системах может быть использована избыточность:

1) временная;

2) алгоритмическая;

3) структурная.

Временная избыточность заключается в наличии дополнительного времени для решения задачи, с тем, чтобы в случае возникновения сбоя или других ошибок можно было исправлять их путем повторения вычислений. Она использует некоторую часть производительности компьютера для контроля исполнения ПО и восстановления вычислительного процесса. Поэтому при проектировании должен быть предусмотрен запас производительности.

Алгоритмическая избыточность заключается в применении таких алгоритмов, которые обеспечивают удовлетворительные результаты в случае наличия или возникновения ошибок в процессе вычислений.

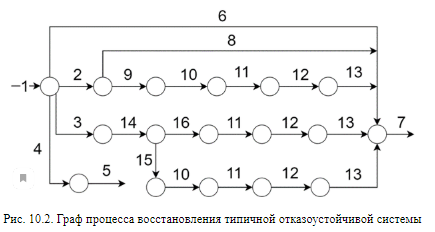
Например, свойствами избыточных алгоритмов обладают итерационные алгоритмы, обеспечивающие сходимость при случайных отклонениях промежуточных результатов. При возникновении ошибки вычислительный процесс занимает больше времени.

Алгоритмическая избыточность и временная избыточность увеличивают устойчивость системы к сбоям.

Структурная избыточность выражается в наличии дополнительных подсистем в структуре компьютерных систем, предназначенных для автоматической замены отказавших элементов, узлов, устройств и подсистем.

Структура пассивно отказоустойчивых систем может основываться, например, на мажоритарном принципе. В общем структура активно отказоустойчивых систем может быть самая различная.

Активная отказоустойчивость может быть применена в многопроцессорных системах с общей памятью, общей шиной, кольцевой, иерархической и другой структурой. Однако процессы восстановления в отказоустойчивых системах имеют много общего. Рассмотрим граф процесса восстановления типичной отказоустойчивой системы (рис.10.2) [23].



На рис. 10.2 ошибка 1 может обнаруживаться либо аппаратными 2 средствами контроля, либо программными 3 средствами контроля, либо не обнаруживаются средствами контроля 4 и в последнем случае результатом является отказ системы 5. В зависимости от степени применения пассивной отказоустойчивости в компьютерных системах ошибка может быть замаскирована 6. В последнем случае вычислительный процесс продолжается без задержки 7.

При обнаружении ошибки аппаратными средствами 2 в большинстве систем проводится повторение выполняемой операции заданное число раз. Если повторение было успешным, т.е. имел место сбой последствия которого при повторении операции исчезли, вычислительный процесс продолжается 8. Для повторения операции необходимо чтобы аппаратные средства сохранили операнды до окончания контроля над выполненной операцией.

Если повторение операции было безуспешным 9, то это говорит об устойчивой ошибке в аппаратуре и поэтому проводится автоматическая реконфигурация 10.

Реконфигурация может заключаться либо в замене отказавшей подсистемы за счет резервов, либо в ее простом отключении. В последнем случае имеет место постепенная деградация системы. После реконфигурации производится восстановление информации 12. Для этого по ходу вычислительного процесса предусматривают контрольные точки 11, в которых состояние системы и вычислительного процесса подвергаются контролю. В случае положительного результата контроля состояние данной программы и данного процессора (промежуточные результаты, содержание регистров и др.) записываются либо в оперативную память другого процессора, либо в общей оперативной памяти, либо на внешнем носителе.

В ходе восстановления информации содержание этих дублирующих записей переписывается в тот процессор, который после реконфигурации берет на себя функции отказавшего. Затем, начиная с контрольной точки вычислительный процесс возобновляется 13.

Аналогичные процедуры проводятся в случае, когда ошибка обнаружена программными средствами 3.Но при этом повторение операции не имеет смысла, так как программные средства обнаруживают отказ с запозданием, за которое были выполнены уже другие операции и поэтому первоначальные операнды вряд ли сохранятся.

Следовательно, после обнаружения ошибки программными средствами могут быть задействованы тесты 14. Если тесты подтверждают наличие устойчивого отказа 15, то следует реконфигурация 10, возврат к контрольной точке 11, восстановление данных 12 и повторение вычислений 13.

Если устойчивого отказа нет 16, то повторяются перечисленные операции без реконфигурации.

Восстановление может оказаться безуспешным также в случае наличия ошибки в программах, разрушения информации в контрольных точках, исчерпания резервов или снижения производительности системы из-за отказов ниже допустимого уровня.

Описанный выше процесс может варьироваться в конкретных системах, особенно, что касается способов обнаружения отказов.

### **Оптимальное распределение ресурсов в отказоустойчивых компьютерных системах.**

### **позволяющих получить уточненное решение, и конечная цель оптимизации: отыскание наилучшего или оптимального решения из всех возможных.**

### **При проектировании отказоустойчивых систем следует стремиться не только к достижению необходимой надежности, но и к достижению этой надежности при минимальных затратах средств.**

### **В отказоустойчивых системах существует ряд параметров, от которых зависит отказоустойчивость системы: X = (x1, …, xn). Этими параметрами могут быть конкретное количество резервных элементов системы, количество версий ПО, временные периоды контроля и восстановления и т. д.**

### **Надежность КС зависит от некоторых параметров монотонно, например, увеличение количества резервов ведет к росту надежности.**

### **Возможны 2 постановки задачи:**

### **1) Нахождение экстремума показателей надежности ПС, произведенное при некоторых ограничениях:**

### **min(max) П(x), где x G**

### **2) Но если целью оптимизации является обеспечение не максимально, а заданной надежности системы, то решается обратная задача:**

### **min(max) Ф(x), где П(х) Н**

### **Методы оптимизации:**

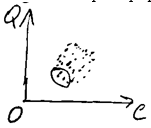
### **- Метод перебора:**

### **Численные методы определения оптимального резерва позволяют найти сколь угодно точное решение и применимы для очень сложных моделей надежности. Простейшим численным методом оптимизации является метод перебора, когда сравниваются между собой все возможные варианты структуры КС, и выбирают тот вариант, который лучше всего отвечает установленным требованиям. Например, структура отказоустойчивой КС имеет вид:**

### 

Представляет собой максимально возможное число параллельных подсистем i-го типа, тогда число конкурирующих вариантов определяется следующим образом: **N =**

Из анализа данной формулы видно, что **число конкурирующих вариантов** возможных решений может быть **очень большим**, и поэтому данный метод может применяться только для **очень простых моделей надежности**. Поэтому для сокращения числа вариантов моделей перебора вводится понятие **доминирующей последовательности**. Рассмотрим график вариантов технических решений:



Из анализа 1-ой и 2-ой постановки задачи видно, что наибольший интерес представляют те решения, которые обладают **минимальной стоимостью и максимальной надежностью**, т. е. область **снизу слева**. Подмножество вариантов этих решений перспективно с точки зрения поиска оптимального варианта решения. **Последовательность таких решений называется доминирующей**. Обычно мощность доминирующей последовательности намного меньше мощности всех вариантов, и поэтому не составляет труда выбрать оптимальное решение из вариантов, входящих в доминирующую последовательность.

Фактически **метод полного перебора** **позволяет найти глобальный экстремум**, но применим **для простых систем**.

- **Градиентный метод поиска локального экстремума**:

Является **простым и достаточно эффективным** методом. В общем случае решение задач градиентным методом заключается в том, что **отыскивается значение** **экстремума целевой функции путем последовательных шагов из начальной точки по направлению градиента**, т. е. максимального увеличения или уменьшения целевой функции. Целевая функция позволяет сравнивать альтернативные решения.

Сначала рассматривается исходная нерезервированная система:

На 1-ом шаге **отыскивается элемент** системы, добавление к которому дает **наибольшее отношение прироста показателя надежности к приросту затрат** (стоимости).

На 2-ом шаге отыскивается **следующий элемент** системы, который даст максимальное отношение прироста надежности к приросту затрат.

В общем градиентный метод позволяет определить **только часть элементов доминирующей последовательности**.

**Итерации** повторяются до тех пор, пока **надежность не достигнет заданной**. Однако результат данного метода **не является самым оптимальным**.

Данный подход может быть применим при оптимизации конфигурации компьютерных сетей. Самые приемлемые решения достигаются наращиванием сети т. о., чтобы прирост надежности, отнесенный к единице расходов был максимальным.м