**1) Фактори якості програмного забезпечення.**

1. Функциональность (Functionality)

* Совместимость в смысле алгоритма
* Точность (соответствие между функцией и требованиями)
* Взаимодействие
* Безопасность

1. Надежность (Reliability)

* Зрелость
* Толерантность к ошибке
* Самовосстановление

1. Удобство использования (Usability)

* Способность к пониманию (understandability)
* Способность к обучению (learnability)
* Работоспособность (operability)
* Привлекательность (attractiveness)

1. Эффективность (Efficiency)

* Временное поведение (time behavior)
* Использование ресурсов

1. Сопровождаемость (Maintainability)

- Анализируемость (Analysability)

- Изменяемость (Changeability)

- Стабильность (Stability)

- Тестируемость (Testability)

6. Мобильность (Portability)

- Адаптируемость (Adaptability)

- Настраиваемость (простота внедрения, Installability)

- Совместимость (Co-existence)

- Взаимозаменяемость (Replaceability)

**ФУНКЦИОНАЛЬНОСТЬ (FUNCTIONALITY)**

Способность программного продукта обеспечивать функции, удовлетворяющие установленные и подразумеваемые потребности при применении программного средства в заданных условиях. Эта характеристика определяет,*что*делает ПС в соответствии с потребностями. Другие характеристики определяют,*когда*и*как*эти потребности удовлетворяются.

***Пригодность (Suitability)***– способность программного продукта обеспечивать набор функций, соответствующий специфическим задачам и целям пользователей.

***Правильность (корректность, Accuracy)***– способность программного продукта обеспечивать правильные или приемлемые результаты или эффекты с необходимой степенью точности.

***Способность к взаимодействию(Interoperability)***– способность про-

граммного продукта взаимодействовать с одной или несколькими заданными системами.

***Защищенность (Security)***– способность программного продукта защищать информацию и данные так, чтобы несанкционированные пользователи или системы не могли прочитать или модифицировать их, а санкционированные пользователи или системы не могли получить отказа в доступе к ним.

**НАДЕЖНОСТЬ (RELIABILITY)**

Способность программного продукта поддерживать заданный уровень качества функционирования при его использовании в заданных условиях. Ограничения надежности в процессе эксплуатации вызваны ошибками в требованиях, проектировании и кодировании.

***Завершенность (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.

***Готовность (доступность, Availability)***– способность программного продукта быть в состоянии выполнять требуемую функцию в данный момент

**ПРАКТИЧНОСТЬ (USABILITY)**

Способность программного продукта быть понятным, изученным, использованным и привлекательным для пользователя при применении в заданных условиях. Очевидно, что практичность зависит от некоторых аспектов таких характеристик качества, как функциональность, надежность и эффективность. Подхарактеристики*Практичности*приведены нарис. 46.

***Понятность (Understandability)***– способность программного продукта, обеспечивающая понимание пользователем пригодности и способа использования программного средства для конкретных задач и условий применения.

***Обучаемость (Learnability)***– способность программного продукта, обеспечивающая изучение пользователем принципов его применения.

***Простота использования (Operability)***– способность программного продукта, позволяющая пользователю эксплуатировать его и управлять . имНа простоту использования влияют некоторые аспекты таких подхарактеристик, как пригодность, изменяемость, адаптируемость и простота внедрения. Простота использования зависит, например, от контролируемости и устойчивости к ошибке.

***Привлекательность (Attractiveness)***– способность программного продукта нравиться пользователю. Данная подхарактеристика связана со свойствами оформления ПП (например использование цветов, графики и т.п.).

**ЭФФЕКТИВНОСТЬ (EFFICIENCY)**

Способность программного продукта обеспечить соответствующую производительность в зависимости от количества используемых вычислительных ресурсов в заданных условиях. Ресурсы могут включать другие программные продукты, конфигурацию программных и аппаратных средств системы и материалы.

***Поведение во времени (Time behaviour)***– способность программного продукта обеспечивать соответствующие времена отклика и обработки, также пропускную способность при выполнении своих функций в заданных условиях.

***Использование ресурсов (Resource utilisation)***– способность программно-го продукта использовать соответствующее количество всех типов ресурсов при выполнении своих функций в заданных условиях. В данной подхарактеристике человеческие ресурсы не учитываются. Они учитываются в такой характеристике качества в использовании, как продуктивность.

***Соответствие эффективности (Efficiency compliance)***– способность программного продукта соответствовать стандартам и соглашениям, связанным с эффективностью.

**СОПРОВОЖДАЕМОСТЬ (MAINTAINABILITY)**

Способность программного продукта к модификации. Модификации могут включать исправления, усовершенствования или адаптацию ПС к изменениям в среде применения, в требованиях и функциональных спецификациях.

***Анализируемость (Analysability)***– способность программного продукта к диагностике его недостатков или причин отказов или к идентификации его частей, которые должны быть модифицированы.

***Изменяемость (Changeability)***– способность программного продукта к реализации заданной модификации. Реализация включает проектирование, кодирование и изменение документации.

***Стабильность (Stability)***– способность программного средства предотвращать непредвиденные эффекты от его модификации.

***Тестируемость (Testability)***– способность программного продукта к проверке результата модификации.

**МОБИЛЬНОСТЬ (PORTABILITY)**

Способность программного продукта к переносу из одной среды в другую. Среда может включать организационное, аппаратное и программное окружение. Подхарактеристики*Мобильности*представлены нарис. 46.

***Адаптируемость (Adaptability)***– способность программного продукта к адаптации к различным окружающим средам без применения дополнительных действий или средств. Адаптируемость включает масштабируемость внутренних возможностей (например, областей экранов, таблиц, форматов отчетов).

***Настраиваемость (простота внедрения, Installability)***– способность программного продукта устанавливаться в заданной среде окружения.

***Совместимость (Co-existence)***– способность программного продукта к сосуществованию с другими независимыми программными средствами в общей среде, разделяя общие ресурсы.

***Взаимозаменяемость (Replaceability)***– способность программного продукта к использованию вместо другого(заданного) ПП с той же целью и в той же среде. Например, для пользователя важна взаимозаменяемость новой версии ПП с его старой версией. Взаимозаменяемость может включать атрибуты таких подхарактеристик, как настраиваемость и адаптируемость.

**2) МЕТРИКИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ХОЛСТЕДА.**

*n1* – число уникальных операторов

*N1* – общее число всех операторов

*n2* – число уникальных операндов

*N2* – общее число всех операндов

*n2'* – кол-во входных и выходных параметров

*n = n1 + n2* – словарь программы

*N = N1 + N2* – длина реализации

*N' = n1 \* log2(n1) + n2 \* log2(n2)* – длина программы (для стилистически корректных программ отклонение *N* от *N'* не превышает 10%)

*V = N \* log2(n)* – объем программы

*V' = (n2' + 2) \* log2(n2' + 2)* – потенциальный объем программы

*L = V' / V* – уровень качества программы (для идеальной программы *L = 1*)

*λ = V'^2 / V = L / V'* – уровень языка программирования

*I = [(2 \* n2) / (n1 \* N2)] \* (N1 + N2) \* log2(n1 + n2) = V / D* – информационное содержание программы (умственные затраты на создание программы)

*E = V^2 / V' = V / L* – необходимые интеллектуальные усилия при разработке программы

*T = E / S* – время составления программы, (число Страуда *S = 18*)

*M = m – n + 2p, p = 1* – метрика Маккейба

*M = n2' / 6* – количество модулей

*B' = V / 3000* – оценка числа дефектов в программе

**Объем программы по Холстеду** определяет объем кода, запрограммированного алгоритма. Размер метрики должен быть от 20 до 1000 для одной функции и от 100 до 8000 для одного файла. Если размер метрики превышает верхнюю границу, то рекомендуется более детально изучить функциональную нагрузку на исследуемый элемент и разбить его на несколько составляющих либо провести оптимизацию алгоритма.

**Операнды**

При расчете метрики Холстеда используются с

ледующие операнды:

*Идентификаторы* – все идентификаторы которые не являются зарезервированными словами;

*Идентификаторы типов* – зарезервированные слова обозначающие тип данных: bool, char, double, float, int, long, short, signed, unsigned, void;

*Константы* – числовые, символьные, строковые константы.

**Операторы**

Ключевые слова следующих категорий, которые интерпретируются как операторы:

Идентификаторы класса памяти: inline, register, static, typedef, virtual, mutable.

Квалификаторы типа: const, friend, volatile.

Зарезервированные слова: asm, break, case, class, continue, default, delete, do, else, enum, for, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const\_cast, static\_cast, dynamic\_cast, reinterpret\_cast, typeid, template, explicit, true, false, typename

Операторы языка программирования:! != % %= &&& || &= ( ) \* \*= + ++ += , - -- -= -> . ... / /= : :: <<<<<= <= = == >>= >>>>= ? [ ] ^ ^= { } | |= ~

Следующие управляющие структуры for (...) if (...) switch (...) while for (...) and catch (...) интерпретируются как один оператор.

Холстед вводит *n'* – **теоретический словарь программы**, т.е. словарный запас, необходимый для написания программы с учетом того, что необходимая функция уже реализована в данном языке и, следовательно, программа сводится к вызову этой функции. Например, согласно Холстеду возможное осуществление процедуры выделения простого числа могло бы выглядеть так:

CALL SIMPLE (X, Y),

где Y – массив численных значений, содержащих искомое число X.

Теоретический словарь в этом случае будет состоять из

*n1'*: {CALL, SIMPLE (...)}, *n1'* =2;

*n2'*: {X,Y}, *n2'* =2;

а его длина, определяемая как *n' = n1' + n2'* будет равна 4.

Для вычисления цикломатического числа Маккейба Z(G) применяется формула

Z(G)=e-v+2p,

где e - число дуг ориентированного графа G;

v - число вершин;

p - число компонентов связности графа.

Число компонентов связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильно связный. Cильно связным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых от точки входа участков и "висячих" точек входа и выхода, сильно связный граф, как правило, получается путем замыкания дугой вершины, обозначающей конец программы, на вершину, обозначающую точку входа в эту программу (p = 1).

**3)Регресійне тестування**

**Регрессионное тестирование** - это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде. Регрессионными могут быть как **функциональные**, так и **нефункциональные тесты**.

Как правило, **для регрессионного тестирования используются тест кейсы, написанные на ранних стадиях разработки и тестирования**. Это дает гарантию того, что изменения в новой версии приложения не повредили уже существующую функциональность. Рекомендуется делать [**автоматизацию регрессионных тестов**](http://www.protesting.ru/automation/), для ускорения последующего процесса тестирования и обнаружения дефектов на ранних стадиях разработки программного обеспечения.

Сам по себе термин "Регрессионное тестирование", в зависимости от контекста использования может иметь разный смысл. Сэм Канер, к примеру, описал 3 основных типа регрессионного тестирования:

* **Регрессия багов** (**Bug regression**) - попытка доказать, что исправленная ошибка на самом деле не исправлена
* **Регрессия старых багов** (**Old bugs regression**) - попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.
* **Регрессия побочного эффекта** (**Side effect regression**) - попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения

**4) Визначення поняття верифікації програмного забезпечення**

Верифікацією й атестацією називаються процеси перевірки й аналізу, у ході яких перевіряється відповідність програмного забезпечення своєї специфікації і вимогам замовників. Верифікація й атестація охоплюють весь цикл життя ПЗ – вони починаються на етапі аналізу вимог і завершуються перевіркою програмного коду на етапі тестування програмної системи. Верифікація відповідає на запитання, чи правильно створена система, а атестація відповідає на запитання, чи правильно працює система. Верифікація перевіряє відповідність ПЗ системній специфікації, зокрема функціональним і нефункціональним вимогам.

У процесах верифікації й атестації використовуються дві основні методики перевірки й аналізу систем: інспектування ПЗ і тестування ПЗ. Інспектування ПЗ має на увазі аналіз і перевірку різних представлень системи, наприклад, документації. Інспектування відбувається на всіх етапах розробки програмної системи. Паралельно з інспектуванням може проводитися автоматичний аналіз вихідного коду програм і відповідних документів. Інспектування й автоматичний аналіз – це статичні методи верифікації й атестації, оскільки їм не потрібна виконуюча система. Тестування – динамічний метод верифікації й атестації, тому що застосовується до системи, що виконується.

У процесі планування верифікації й атестації необхідно визначити співвідношення між статичними і динамічними методами перевірки системи, визначити стандарти і процедури інспектування, скласти план тестування програм. Від типу розроблювальної системи залежить те, чому варто приділити більше уваги – інспектуванню або тестуванню. Чим більше критична система, тим більше уваги варто приділяти статичним методам верифікації. Отже, процеси верифікації й атестації інспектування і тестування варто використовувати спільно.

**5) Метрики якості програмного забезпечення маккейба**

Основанная на анализе передачи управления от 1-го операнда к 2-ому.

Она также была разработана для структурного программирования. Маккейб предельно описал программный код в виде графа где вершины являются операторами, а дугами – переходы между ними.

М = m-n+2 , где m- кол-во ребр, n – к-во вершин.

Цикломатическое число Маккейба: z = m-n+2p, p=1

Ц. число показывает его полносвязность (сколько цыклов)

Р – кол-во ребр, необходимых чтобы граф был полносвязным.

Ц. число Маккейба показывает сколько тестов надо запустить. Чтобы полностью протестировать программный код по одному з критериев покрытия. Если z>10, программа обладает излишней сложностью, и его следует разбить на модули.

Существует упрощённая формула расчета z:

М = u+1, u – к-во операторов условий

**6) Обнаружение ошибок**

Большинство методов направлено по возможности на неза­медлительное обнаружение сбоев. Немедленное обнаружение имеет два преимущества: можно минимизировать влияние ошиб­ки и последующие затруднения для человека, которому придется извлекать информацию о ней, находить ее и исправлять.

{SITELINK-S405}Меры по обнаружению ошибок {/SITELINK}можно разбить на две под­группы: ***пассивные***попытки обнаружить симптомы ошибки в про­цессе «обычной» работы программного обеспечения и ***активные***попытки программной системы периодически обследовать свое состояние в поисках признаков ошибок.

***Пассивное обнаружение****.*Меры по обнаружению ошибок могут быть приняты на нескольких структурных уровнях программной системы. Здесь мы будем рассматривать уровень подсистем, или ком­понентов, т.е. нас будут интересовать меры по обнаружению симп­томов ошибок, предпринимаемые при переходе от одного компо­нента к другому, а также внутри компонента. Все это, конечно, при­менимо также к отдельным модулям внутри компонента.

Разрабатывая эти меры, мы будем опираться на следующее.

1. *Взаимное недоверие.*Каждый из компонентов должен пред­полагать, что все другие содержат ошибки. Когда он получает какие-нибудь данные от другого компонента или из источника вне системы, он должен предполагать, что данные могут быть неправильными, и пытаться найти в них ошибки.

*2. Немедленное обнаружение.*Ошибки необходимо обнаружить как можно раньше. Это не только ограничивает наносимый ими ущерб, но и значительно упрощает задачу отладки.

3. *Избыточность.*Все средства обнаружения ошибок основаны на некоторой форме избыточности (явной или неявной).

*Активное обнаружение ошибок.*Не все ошибки можно выя­вить пассивными методами, поскольку эти методы обнаружива­ют ошибку лишь тогда, когда ее симптомы подвергаются соот­ветствующей проверке. Можно делать и дополнительные провер­ки, если спроектировать специальные программные средства для активного поиска признаков ошибок в системе. Такие средства называются *средствами активного обнаружения ошибок.*

Активные средства обнаружения ошибок обычно объединя­ются в *диагностический монитор:*параллельный процесс, кото­рый периодически анализирует состояние системы с целью обна­ружить ошибку. Большие программные системы, управляющие ресурсами, часто содержат ошибки, приводящие к потере ресур­сов на длительное время. Например, управление памятью опера­ционной системы сдает блоки памяти «в аренду» программам пользователей и другим частям операционной системы. Ошибка в этих самых «других частях» системы может иногда вести к не­правильной работе блока управления памятью, занимающегося возвратом сданной ранее в аренду памяти, что вызывает медлен­ное вырождение системы.

**7) Концепция тестирования**

*Программа* – это аналог формулы в обычной математике.

Формула для функции f, полученной *суперпозицией* функций f1, f2, ... fn – *выражение*, описывающее эту *суперпозицию*.

f = f1\* f2\* f3\*... \* fn

Если аналог f1,f2,... fn – *операторы* языка программирования, то их формула – *программа*.

Существует два метода обоснования *истинности формул*:

1. **Формальный подход** или **доказательство** применяется, когда из исходных формул-аксиом с помощью формальных процедур (правил вывода) выводятся искомые формулы и утверждения (теоремы). Вывод осуществляется путем перехода от одних формул к другим по строгим правилам, которые позволяют свести процедуру перехода от формулы к формуле к последовательности текстовых подстановок:
2. A\*\*3 = A\*A\*A

A\*A\*A = A -> R, A\*R -> R, A\*R -> R

Преимущество формального подхода заключается в том, что с его помощью удается избегать обращений к бесконечной области значений и на каждом шаге доказательства оперировать только конечным множеством символов.

1. **Интерпретационный подход** применяется, когда осуществляется подстановка констант в формулы, а затем интерпретация формул как осмысленных утверждений в элементах множеств конкретных значений. *Истинность* интерпретируемых формул проверяется на конечных множествах возможных значений. Сложность подхода состоит в том, что на конечных множествах комбинации возможных значений для реализации исчерпывающей проверки могут оказаться достаточно велики.

Интерпретационный подход используется при экспериментальной проверке соответствия программы своей спецификации

Применение интерпретационного подхода в форме экспериментов над исполняемой программой составляет суть *отладки* и *тестирования*.

**Основная терминология**

***Отладка (debug, debugging)*** – процесс поиска, локализации и исправления ошибок в программе [[ 9 ]](https://www.intuit.ru/studies/courses/48/48/literature#literature.9)[IEEE Std.610-12.1990].

Термин " *отладка* " в отечественной литературе используется двояко: для обозначения активности по поиску ошибок (собственно *тестирование*), по нахождению причин их появления и исправлению, или активности по локализации и исправлению ошибок.

***Тестирование*** обеспечивает выявление (констатацию наличия) фактов расхождений с требованиями (ошибок).

Как правило, на ***фазе тестирования*** осуществляется и исправление идентифицированных ошибок, включающее локализацию ошибок, нахождение причин ошибок и соответствующую корректировку программы тестируемого приложения (Application Under Testing (AUT) или Implementation Under Testing (IUT)).

Если *программа* не содержит синтаксических ошибок (прошла трансляцию) и может быть выполнена на компьютере, она обязательно вычисляет какую-либо функцию, осуществляющую *отображение* входных данных в выходные. Это означает, что *компьютер* на своих ресурсах доопределяет частично определенную программой функцию до тотальной определенности. Следовательно, судить о правильности или неправильности результатов выполнения программы можно, только сравнивая спецификацию желаемой функции с результатами ее вычисления, что и осуществляется в процессе *тестирования*.

**8)** **Зв’язок задач валідації, верифікації та тестування с життевим циклом програмного забезпечення.**

**Валідація програмного забезпечення** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *Validation Software*) — процес визначення відповідності розроблюваного [програмного забезпечення](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) між очікуваннями і потребами користувача, вимогам до системи. Валідація є одним із основних етапів [тестування програмного забезпечення](https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F).

Мета процесу валідації — переконатися, що специфічні вимоги для програмного продукту виконано, і здійснюється це за допомогою:

* розробленої стратегії і критеріїв перевірки всіх робочих продуктів;
* обговорених дій з проведення валідації;
* демонстрації відповідності розроблених програмних продуктів вимогам замовника і правилам їхнього використання;
* узгодження із замовником отриманих результатів валідації продукту.

Процес валідації може проводитися самим виконавцем або іншою особою, наприклад, замовником, що здійснює дії з впровадженню і проведенню цього процесу за планом, у якому відбиті елементи і задачі перевірки. При цьому використовуються методи, інструментальні засоби і процедури виконання задач процесу для встановлення відповідності тестових вимог і особливостей використання програмних продуктів проекту на правильність реалізації вимог.

[Верифікація](https://uk.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) і валідація полягає в перевірці специфікацій і правильності виконання програм відповідно до заданих вимог і формального опису програми.

Верифікація програмного коду допомагає зробити висновок про коректність створеної програмної системи при її проектуванні і після завершення її розроблення. Валідація дозволяє встановити здійснимість заданих вимог шляхом їх перегляду, інспекції і оцінки результатів проектування на процесах [життєвого циклу](https://uk.wikipedia.org/wiki/%D0%96%D0%B8%D1%82%D1%82%D1%94%D0%B2%D0%B8%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) для підтвердження того, що здійснюється коректна реалізація вимог, дотримання заданих умов і обмежень до системи. Верифікація і валідація забезпечують перевірку повноти, несуперечності і однозначності специфікації і правильності виконання функцій системи.

[Верифікації](https://uk.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) і валідації піддаються:

* компоненти системи, їх інтерфейси (програмні, технічні і інформаційні) і взаємодія об'єктів (протоколи, повідомлення) у розподілених середовищах;
* описи доступу до [баз даних](https://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85), засоби захисту від несанкціонованого доступу до даних різних користувачів;
* документація до системи;
* тести, тестові процедури і вхідні набори даних.

На інших процесах [життєвого циклу](https://uk.wikipedia.org/wiki/%D0%96%D0%B8%D1%82%D1%82%D1%94%D0%B2%D0%B8%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) виконуються додаткові дії:

* перевірка і контроль проектних рішень за допомогою методик і процедур перегляду ходу розроблення;<
* звернення до CASE-систем , що містять у собі процедури перевірки вимог до продукту;
* перегляди й інспекції проміжних результатів на відповідність вимогам для підтвердження того, що програмна система має коректну

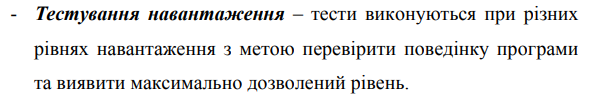
реалізацію вимог і задовольняє умови виконання.

Таким чином, основні задачі процесів верифікації і валідації полягають у тому, щоб перевірити і підтвердити, що кінцевий програмний продукт відповідає призначенню і задовольняє вимогам замовника. Ці процеси взаємозалежні і визначаються, як правило, одним загальним терміном «верифікація і валідація» або «Verification and Validation» (V&V).

V&V засновані на плануванні їх як процесів, так і перевірки для найбільш критичних елементів проекту: компонентів, інтерфейсів (програмних, технічних і інформаційних), взаємодій об'єктів (протоколів і повідомлень), передачі даних між компонентами і їхнього захисту, а також створення тестів і тестових процедур.

Після перевірки окремих компонентів системи проводяться їхня інтеграція, повторна верифікація і валідація інтегрованої системи, створюється комплект документації, що відображає правильність виконання вимог за результатами інспекцій і [тестування тестування](https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F).

**9)**  **Принципи тестування навантаженням**



# **Нагрузочное тестирование**

**Нагрузочное тестирование** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *load testing*) — подвид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Основная цель нагрузочного тестирования заключается в том, чтобы, создав определённую ожидаемую в системе нагрузку (например, посредством виртуальных пользователей) и, обычно, использовав идентичное программное и аппаратное обеспечение, наблюдать за показателями производительности системы.

|  |
| --- |
| ***Пример 1:***  *Веб-сервис с функциональностью корзины покупателя рассчитан на 100 одновременно работающих пользователей, которые следуют некоторому определённому сценарию (заданные действия в указанных пропорциях):*   * *25 пользователей просматривают товар и выходят из системы.* * *25 пользователей добавляют товар в корзину, оформляют его и выходят из системы.* * *25 пользователей используют функцию возврата товара и выходят из системы.* * *25 пользователей входят в систему и не проявляют никакой активности.*   *В данном случае нагрузочное тестирование должно эмулировать вышеописанный типичный сценарий работы с веб-сервисом с целью удостовериться, что система готова к выходу в эксплуатацию. При этом для анализа могут сниматься показатели производительности системы в целом или каждого узла системы в частности.* |

### Основные принципы нагрузочного тестирования

**1. Уникальность запросов**

***Даже сформировав реалистичный сценарий работы с системой на основе статистики её использования, необходимо понимать, что всегда найдутся исключения из этого сценария.***

В случае *Примера 1* это может быть пользователь, обращающийся к отличным от всех остальных, уникальным страницам веб-сервиса.

**2. Время отклика системы**

***В общем случае время отклика системы подчиняется***[***функции нормального распределения***](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5)***.***

В частности, это означает, что, имея достаточное количество измерений, можно определить вероятность с которой отклик системы на запрос попадёт в тот или иной интервал времени.

**3. Зависимость времени отклика системы от степени распределённости этой системы.**

[***Дисперсия***](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BF%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D0%BE%D0%B9_%D0%B2%D0%B5%D0%BB%D0%B8%D1%87%D0%B8%D0%BD%D1%8B)***нормального распределения времени отклика системы на запрос пропорциональна отношению количества узлов системы, параллельно обрабатывающих такие запросы и количеству запросов, приходящихся на каждый узел.***

То есть, на разброс значений времени отклика системы влияет одновременно количество запросов приходящихся на каждый узел системы и само количество узлов, каждый из которых добавляет некоторую случайную величину задержки при обработке запросов.

**4. Разброс времени отклика системы**

***Из утверждений 1, 2 и 3 можно также заключить, что при достаточно большом количестве измерений величины времени обработки запроса в любой системе всегда найдутся запросы, время обработки которых превышает определённые в требованиях максимумы; причем, чем больше суммарное время проведения эксперимента тем выше окажутся новые максимумы.***

Этот факт необходимо учитывать при формировании требований к производительности системы, а также при проведении регулярного нагрузочного тестирования.

**5. Точность воспроизведения профилей нагрузки**

***Необходимая точность воспроизведения профилей нагрузки тем дороже, чем больше компонент содержит система.***

Часто невозможно учесть все аспекты профиля нагрузки для сложных систем, так как чем сложнее система, тем больше времени будет затрачено на проектирование, программирование и поддержку адекватного профиля нагрузки для неё, что не всегда является необходимостью. Оптимальный подход в данном случае заключается в балансировании между стоимостью разработки теста и покрытием функциональности системы, в результате которого появляются допущения о влиянии на общую производительность той или иной части тестируемой системы.

**10) Стадии тестирования в процессе разработки программного обеспечения.**

**Стадии разработки ПО** — это этапы, которые проходят команды разработчиков ПО, прежде чем программа станет доступной для широко круга пользователей.

**Тестирование** – вид деятельности, целью которого является проверка соответствия программного продукта возможностям, заданным в техническом задании и описанным разработчиком.

- В жизненном цикле программного обеспечения (ПО) процесс тестирования в основном стоит за этапом разработки, перед эксплуатацией продукта пользователями.

- С точки зрения данных исследований тестирование может быть осуществлено в различные стадии разработки, а не только на завершающем этапе и применено не только к программному коду.

**Виды (уровни) тестирований: приемочное, системное, интеграционное и модульное.**

**Модульное.**

Когда программист написал отдельный модуль, который выполняет определенные функции, его отдают тестировщику, который проверяет правильность работы этого модуля, это производится после написания каждого отдельного модуля.

**Интеграционное.**

Переходной этап между модульным и системным, в котором на соответствие требований проверяется интеграция модулей, их взаимодействие между собой, а также интеграция подсистем в одну общую систему. Для интеграционного тестирования используются компоненты, уже проверенные с помощью модульного тестирования, которые группируются в множества. Данные множества проверяются в соответствии с планом тестирования, составленным для них, а объединяются они через свои интерфейсы.

**Системное.**

тестирование выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям, как функциональным, так и не функциональным. Выполняя системное тестирование, можно обнаружить следующие типы дефектов: неправильное использование системных ресурсов, непредусмотренные комбинации пользовательских данных, проблемы с совместимостью окружения

**Приемочное.**

Вид тестирования, проводимый на этапе сдачи готового продукта (или готовой части продукта) заказчику. Целью приемочного тестирования является определение готовности продукта, что достигается путем прохода тестовых сценариев и случаев, которые построены на основе спецификации требований к разрабатываемому ПО.

Результатом приемочного тестирования может стать:

• Отправка проекта на доработку.

• Принятие его заказчиком, в качестве выполненной задачи.

**11) Модель управління якістю програмного забезпечення - CMMI.**

Capability Maturity Model Integration (CMMI) — це підхід для вдосконалення процесів, який забезпечує організації суттєвими елементами ефективних процесів. Він може використовуватись для покращення процесу як на рівні проекту чи відділу, так і на рівні цілої організації. CMMI дозволяє інтегрувати традиційно відокремлені організаційні функції.

CMMI визначає 22 процесні області . Для кожної із процесних областей існує ряд цілей (goals), які повинні бути досягнуті при впровадженні CMMI в даній конкретній процесній області. Деякі цілі є унікальними — вони називаються спеціальними. Загальні цілі застосовуються одразу до декількох процесних областей. Цілі досягаються за допомогою практик; так само, як і цілі, практики діляться на спеціальні та загальні.

Існує два представлення CMMI — поетапне і неперервне .Поетапне представлення групує процесні області у п'ять рівнів *зрілості*. Неперервне представлення визначає рівні *можливостей*.

5 рівнів : **1. Початковий** . Якщо організаця знаходиться на цьому рівні то ключові області процесів для неї не передбачено

**2. Повторюваний .** Управління програмними конфігураціями . Забезпечення якості програмних продуктів . Управління контрактами підрядчиків.Контроль за ходом проектів. Планування програмних проектів . Управління вимогами

**3.Визначений .** Експертні оцінки. Координація взаємодії проектних груп. Інженерія програмного продукту. Комплексний менеджмент ПЗ. Програма навчання персоналу. Визначення організаційного процесу.

**4 Керований .** Менеджмент якості ПЗ. Управління процессом на основі кількісних методів.

**5 Опитмізований.** Управління змінами в процесі . Управління технологічними змінами . Запобігання дефектів

**12) Інтеграційне тестування**

**Інтеграційне тестування** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *integration testing*) — це фаза [тестування програмного забезпечення](https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F), під час якої окремі модулі програми комбінуються та тестуються разом, у взаємодії. Інтеграційне тестування виконується після [модульного тестування](https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) та перед [верифікацією](https://uk.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) та [валідацією](https://uk.wikipedia.org/wiki/%D0%92%D0%B0%D0%BB%D1%96%D0%B4%D0%B0%D1%86%D1%96%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) [ПЗ](https://uk.wikipedia.org/wiki/%D0%9F%D0%97). Якщо розглядати цей процес як [систему](https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), то на вхід їй подаються модулі, які вже пройшли модульне тестування; потім модулі групуються в більші частини, виконуються тести передбачені планом, а на виході системи — інтегрована система, що готова до [системного тестування](https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F).

Метою інтеграційного тестування є [верифікувати](https://uk.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F) вимоги з функціональності, продуктивності, надійності до основних компонентів програми. Ці компоненти, тобто групи модулів, тестуються методом чорної скриньки ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *«black-box testing»*), успішні та неуспішні тест-кейси [симулюються](https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D1%96%D1%8F) відповідними вхідними параметрами.

В процесі інтеграційного тестування тестуються симульоване використання спільних ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *«shared»*) даних та комунікація між [процесами](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Тест-кейси перевіряють чи коректно взаємодіють всі компоненти, наприклад: через виклик процедури або активізацію процесу.

Дуже важливо те, що до інтеграційного тестування приступають тільки після модульного. Це дозволяє реалізувати стратегію «будівельних блоків», коли до верифікованої системи інтегруються верифіковані модулі.

Серед різновидів інтеграційного тестування є: «big bang» тестування, тестування зверху-донизу та знизу-вгору.

13) Основні проблеми процесу тестування програмного забезпечення.

Тестирование – процесс выполнения действий программы с целью обнаружения в ней ошибок.

Тестовая ситуация – входы для проверки системы и соответственно существ выходы (спецификации требования).

Ошибка – действие программного кода на этапе разработки, приводящее к тому, что ПО работает неправильно, т.е. не соответствует требованиям.

Отказ – непредсказуемое поведение ПО, как правило является ошибкой аналитика, но программист так же может быть причастен.

**14) Модульне тестування**

Тестирование програмного кода на уровне отдельно взятого модуля , функции или классы. Основная цель – найти ошибку в алгоритме. Модульное тестирование проводится по принципе «Белого ящика» . Чтобы реализовать , необходимо создать соответсвующее окружение . Окружение включает в себя заглушки всех интерфейсов из одного модуля. Заглушка – генератор случаных чисел либо набор констант выхода или входа . Стратегия тестирования формируется с направлением входных данных либо выходных. Чтобы реализовать модульное тестирование необходимо реализовать цикл предотвращения дефектов.У нас есть база ошибок и контроль версий данного модуля , на практике это две системы контроль версий и back tracking . Благодаря наличию базы есть возможность анализировать ошибки и источники их возникновения. Это важно для поиска . Модульное тестирование в 99 случаях это структурное тестирование следовательно мы приминяем структурный критерий тестирования . Дополнением к структурному тестированию выступает анализ метрик третей фазей модульного тестирования может быть полуавтоматическое тестирование.

Вторая фаза модульного тестирования ориентирована на тесты алгоритмов. На первом этапе вычисляется количество тестовых путей необходимо выбрать необходимое количество тестовых путей .

**15) Тестирование методом „белій ящик”**

Тестирование методом белого ящика (также: прозрачного, открытого, стеклянного ящика; основанное на коде или структурное тестирование) – метод тестирования программного обеспечения, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику. Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутренне устройство системы, за пределы ее внешних интерфейсов.

**Тестирование белого ящика – это:**

– тестирование, основанное на анализе внутренней структуры компонента или системы.

– тест-дизайн, основанный на технике белого ящика – процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.

Почему «белый ящик»? Тестируемая программа для тестировщика – прозрачный ящик, содержимое которого он прекрасно видит.

Методы белого ящика :

2.1.1 Метод покрытия операторов

Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз.

2.2 Метод покрытия решений (покрытия переходов)

Более сильный метод тестирования известен как покрытие решений (покрытие переходов). Согласно данному методу каждое направление перехода должно быть реализовано по крайней мере один раз.

2.3 Метод покрытия условий-

Лучшие результаты по сравнению с предыдущими может дать метод покрытия условий. В этом случае записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз

2.4 Критерий решений (условий)

Критерий покрытия решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз, все результаты каждого решения

2.5 Метод комбинаторного покрытия условий.

Критерием, который решает эти и некоторые другие проблемы, является комбинаторное покрытие условий. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз. Набор тестов, удовлетворяющих критерию комбинаторного покрытия условий, удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

**16) Надійність програмного забезпечення.**

Надежность программного обеспечения существенно отличается от надежности апаратури.Програмы не изнашиваются, поломка программы невозможна.

О надежности программного обеспечения речь идет только с точки зрения ошибок в нем самом. Совершенно не затронута важная тема использования программ для исправления ошибок аппаратуры или их последствий Важной характеристикой надёжности програмного обеспечения является его восстанавливаемость, которая определяется затратами времени и труда на устранение отказа через проявилась ошибки в программе и его последствий. Восстановление после отказа в программе может заключаться в корректировке восстановлении текста программы, исправлении данных, внесение изменений в организацию вычислительного процесса, часто оказывается необходимым при работе ЭВМ в реальном масштабе времени

Организационные методы обеспечения надежности программного обеспечения включают также моделирование процесса разработки и достижимых уровней надежности. Основным показателем надежности является вероятность того, что оно работает без ошибок в течение фиксированного отрезка времени.

**17) Системное тестирование (System Testing)**

Основной задачей системного тестирования является проверка [как функциональных, так и не функциональных требований](http://www.protesting.ru/testing/testtypes.html) в системе в целом. При этом выявляются [дефекты](http://www.protesting.ru/testing/bugreport.html), такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи.

Можно выделить два подхода к системному тестированию:

на базе требований (requirements based)

Для каждого требования пишутся [тестовые случаи (test cases)](http://www.protesting.ru/testing/testcase.html), проверяющие выполнение данного требования.

на базе случаев использования (use case based)

На основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся [тест кейсы (test cases)](http://www.protesting.ru/testing/testcase.html), которые должны быть протестированы.

18)  Модель комплексного управління якістю програмного забезпечення (на базі ISO).

Комплексное решение задачи обеспечения качества программных систем предполагает разработку и внедрение той или иной системы управления качеством. На данный момент наиболее распространена и используется многоуровневая модель качества программного обеспечения, представленная в наборе стандартов ISO 9126.  
Рассмотрим построение модели качества информационной системы в виде 4-уровневой иерархической структуры показателей качества.

Уровнями модели будут выступать.  
• Факторы – первый уровень.  
• Характеристики – второй уровень.  
• Метрики – третий уровень.  
• Оценочные элементы – четвертый уровень.

В простейшем варианте модели качества выделим таких три основных фактора, которые принадлежат самому верхнему уровню в иерархии структуры показателей качества:  
• Программа и данные.  
• Документация пользователя.  
• Описание продукта.

Рассмотрим формирование требований собственно к программам и данным. По стандарту ISO/IEC 9126 необходимо определить свойства программного обеспечения по каждой из шести характеристик, а именно:  
• Функциональность.  
• Надежность.  
• Используемость (удобство использования).  
• Эффективность.  
• Сопровождаемость.  
• Мобильность.

Для каждой из шести характеристик необходимо обосновать и выбрать полный перечень метрик для построения модели качества системы. Например, характеристика “Функциональность”может быть описана такими метриками:  
• Инсталляция.  
• Наличие функций.  
• Корректность.  
• Совместимость внутренняя.  
• Совместимость внешняя.

Для каждой из метрик необходимо обосновать и выбрать полный перечень оценочных элементов. Например, метрика “Инсталляция” может быть описана следующими оценочными элементами:  
• Наличие возможности выполнить инсталляцию продукта пользователем.  
• Достаточность информации в инструкции по инсталляции для успешной установки продукта.  
• Наличие возможности автозапуска процесса инсталляции (autorun). и т.д.



\*подробнее в вопросе 1\*

**20) Роль керівника проекту при використанні системи відстеження помилок**

Система отслеживания ошибок – это система, ядром которой выступает отчет об ошибках.

Система отслеживания ошибок (англ. bug tracking system) — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения (программистам, тестировщикам и др.) учитывать и контролировать ошибки (баги), найденные в программах, а также следить за процессом устранения этих ошибок.

Задачи системы:

* Отследить ход выполнения работ;
* Организовать взаимодействие между сотрудниками в рамках выполнения проекта;
* Отразить производительность работы;
* Разрешение конфликтов между участниками проекта.

Эта система так же важна руководителю проекта, потому что он может:

1. Управлять временем;
2. Разрешать конфликты;
3. Оценивать производительность.

Руководитель проета планирует и контролирует работы, связанные с тестированием: оценки сроков, работу над планом-графиком, контроль покрытия требований тестами, постановку задач членам команды, коммуникацию со стейкхолдерами.

21) Характеристика хорошего теста

Критерии хорошего теста:

1. Тест должен на основе заданной вероятности выявить ошибку
2. Набор тестов не должен быть избыточным
3. Тест не должен быть не сложным, не простым
4. Тест должен быть лучшим в своей категории

22)  Модель вимірювання характеристик якості програмного забезпечення.

Качество – это характе-ка кот поддается воздействию.

Характерика качества – отображает св-ва ПО по кот можно судить что оно качественое(по кот на него можно воздействовать).

Поскольку очень сложно измерить,доказать качество то вводят иерархические системы измерения качества.

Каждый Программный Продукт описывается своей уникальной системой качества, а представление модели ISO явл общей.

Фактор качества – это свойства в той или иной степени описывающие качество ПО.

При оценке качесттва учитывается 2,3 фактораю Для численой оценки качества используют 1 или несколько критериев.

Критерий качества – это вычислимое выражение кот устанавливает растояния(интервал) по кот мы можем сделать вывод о качестве По с точки зрения заданого фактора.

23) Понятие классов эквивалентности  
Класс эквивалентности - множество тестов со сходными параметрами, протестировав один из них, можно поставить галочку, что протестировал и все остальные (остальные параметры множества будут иметь тот же результат).

Эквивалентные тесты — это тесты, которые приводят к одному и тому же результату.

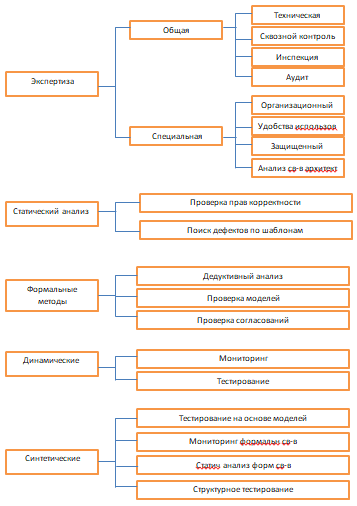
То есть, если мы выполним два любых теста из одного класса эквивалентности — то получим один и тот же результат.

Классы эквивалентности призваны сократить количество тестов и сохранить приемлимое качество покрытия.

*Признаки эквивалентности тестов*:

* направлены на поиск одной и той же ошибки;
* если один из тестов обнаруживает ошибку, другие скорее всего, тоже её обнаружат;
* если один из тестов не обнаруживает ошибку, другие, скорее всего, тоже её не обнаружат;
* тесты используют схожие наборы входных данных;
* для выполнения тестов мы совершаем одни и те же операции;
* тесты генерируют одинаковые выходные данные или приводят приложение в одно и то же состояние;
* все тесты приводят к срабатыванию одного и того же блока обработки ошибок;
* ни один из тестов не приводит к срабатыванию блока обработки ошибок.

24)  Класифікація методів верифікації.



25)  Мутаційні критерії вибору тестів.

В основе их применения – будем считать, что программист пишет с 1го раза почти правильный программный код. Назовем мутациями – мелкие ошибки. Мутанты – те программы, которые отличаются мутациями. Метод означает, что в разрабатываемой программном коде Р, намеренно вносят мутации, формируя набор программного кода с ошибками Р1, …,Рm. Если вес тестовые наборы заблокировали копии программного кода, то программа Р – протестирована.

26)  Основні проблеми процесу тестування програмного забезпечення.

1. Верна ли версия компонент, используемая в программном продукте?
2. Доступны ли все файлы данных?
3. Являются ли корректными ссылки на эти файлы данных?
4. Сама система собр на правильн версии платформы, библиотеки и т.д.?
5. Правильная ли версия компилятора и сопутствующего программного обеспечения?

27)  Ролі в процесі веріфікації програмного забезпечення.

1 роль – Заказчик : имеет право менять требования лишь во взаимодействии с менеджером или лидером проекта.

2 роль – Менеджер проекта : обеспечивает коммуникационный канал между заказчиком и командой. Основная обязанность – получить требования.

3 роль – Менеджер ПО : управляет командой внутри проекта (коммуникатор). Отвечает за разработку функциональных спецификаций. Может их менять, распределять разработчикам задачи и отчитыв за этапы.

4 роль – Разработчик : исходный прог код

5 роль – Специалист по тестированию : разработка тест планов по тестированию.

6 роль – Специалист по контр качества: отвечает за альфа и бета тестирование.

7 роль – Специалист по верификации.

8 роль – Специалист по внедрению и сопровождению.

9 роль – Специалист по безопасности.

10 роль – Технический писатель.

28)  Кількісні характеристики програмного забезпечення та його надійності.

**1) Вероятность безотказной работы***http://ok-t.ru/studopedia/baza14/101510417807.files/image156.jpg —*это вероятность того, что в пределах заданной наработки отказ системы не возникает.

http://ok-t.ru/studopedia/baza14/101510417807.files/image158.jpg

где http://ok-t.ru/studopedia/baza14/101510417807.files/image160.jpg - случайное время работы программно обес­печения до отказа; http://ok-t.ru/studopedia/baza14/101510417807.files/image162.jpg - заданная наработка.

**2) Вероятность отказа**— вероятность того, что в пределах заданной наработки отказ программной системы возникает.

http://ok-t.ru/studopedia/baza14/101510417807.files/image164.jpg

**3) Интенсивность отказов ПО системы***http://ok-t.ru/studopedia/baza14/101510417807.files/image166.jpg*— это условная плот­ность вероятности возникновения отказа программного обеспечения в определенный момент времени при усло­вии, что до этого времени отказ не возник.

http://ok-t.ru/studopedia/baza14/101510417807.files/image168.jpg

где http://ok-t.ru/studopedia/baza14/101510417807.files/image170.jpg — плотность вероятности отказа в момент времени *t.*

http://ok-t.ru/studopedia/baza14/101510417807.files/image172.jpg

Связь между http://ok-t.ru/studopedia/baza14/101510417807.files/image174.jpg определяется следующим соотношением:

http://ok-t.ru/studopedia/baza14/101510417807.files/image176.jpg

**Средняя наработка до отказа** http://ok-t.ru/studopedia/baza14/101510417807.files/image178.jpg — это математическое ожидание време­ни работы программного обеспечения АСОИУ до очередного отказа.

http://ok-t.ru/studopedia/baza14/101510417807.files/image180.jpg

где http://ok-t.ru/studopedia/baza14/101510417807.files/image182.jpg — время работы программного или информационного обеспечения http://ok-t.ru/studopedia/baza14/101510417807.files/image184.jpg

http://ok-t.ru/studopedia/baza14/101510417807.files/image186.jpg

где http://ok-t.ru/studopedia/baza14/101510417807.files/image188.jpg — время работы программного средства между отказами; i=1,2,..n – количество отказов.

**Среднее время восстановления** http://ok-t.ru/studopedia/baza14/101510417807.files/image190.jpg – это математическое ожидание времени восстановления http://ok-t.ru/studopedia/baza14/101510417807.files/image192.jpg , времени, затраченного на восстанов­ление и локализацию отказа http://ok-t.ru/studopedia/baza14/101510417807.files/image194.jpg , времени устранения отказа http://ok-t.ru/studopedia/baza14/101510417807.files/image196.jpg , времени пропускной проверки работоспособности http://ok-t.ru/studopedia/baza14/101510417807.files/image198.jpg :

http://ok-t.ru/studopedia/baza14/101510417807.files/image200.jpg

**Коэффициент готовности***http://ok-t.ru/studopedia/baza14/101510417807.files/image202.jpg*оценивается как вероятность того, что программное и информационное обеспечение АСОИУ будет нахо­диться в работоспособном состоянии в произвольный момент вре­мени его использования по назначению:

http://ok-t.ru/studopedia/baza14/101510417807.files/image204.jpg

29) Функціональні критерії вибору тестів.

Ф.К. – контролируют степень выполнения требований закащика к ПО. Ф.К. работает по модели *черного ящика*( неизвестно что внутри, манипуляции только со входами и выходами):

1. Тестирование пунктов спецификации( созд. Тест кот. Проверяет каждый пункт спецификации хотя бы 1 раз).
2. Тест-ние классов входных данных (проверить каждый класс не менее 1 раза).
3. Разработка тестовых наборов на осн классов эквивалентности.
4. Тестирование правил (в качестве правила рассматриваем соотв-е входных и выходных правил либо вводится спец-я грамматика).
5. Тест-е классов выходных данных.
6. Тест-е функций (это самый популярный критерий по кот мы создаем тестовые наборы, чтобы проверить хотя бы 1 раз любую из функций представленных на UML-диаграмме).

30) Класифікація програмних помилок.

Понятие надежности ПО базируется на понятии ошибка, дефект, сбой.

Отказ - это отклонение программы от функционирования или невозможность программы выполнять функции, определенные требованиями и ограничениями.

Сбой – кратковременный отказ.

Дефект – долговременный отказ.

Ошибка – 1) в ПО это неверный, неправильный перевод из одного представление в другое.

Суть представления – любая программа это трансформатор из исходных данных в понятие. Трансформация выполняется в 3 этапы 1) читающий механизм 2) печать 3) перенос информации.

Ошибка – 2) это искажение, неправильный перевод из 1 представление в др.

Сбой – 2) как правило возникает по причине аппаратного обеспечения.

Дефект – 2) следствие ошибки или группы ошибок.

**31) Призначення та основні компоненти звіту про помилку**

**КРАТКО**

**Дефект** (он же баг) – это несоответствие фактического результата выполнения программы ожидаемому результату. Дефекты обнаруживаются на этапе тестирования программного обеспечения (ПО), когда тестировщик проводит сравнение полученных результатов работы программы (компонента или дизайна) с ожидаемым результатом, описанным в спецификации требований. Итак, как только мы обнаруживаем баг, нам необходимо его задокументировать для продолжения жизненного цикла дефекта (который мы рассматривали ранее).  Документ, который описывает баг, называется – баг репорт.

**Баг репорт (bugreport)** – это технический документ, который содержит в себе полное описание бага, включающее информацию, как о самом баге (короткое описание, серьезность, приоритет и т.д.), так и о условиях возникновения данного бага. Баг репорт должен содержать правильную, единую терминологию, описывающую элементы пользовательского интерфейса и события данных элементов, приводящих к возникновению бага.

В общем случае, баг репорт состоит из:

**Шапка.**

• Короткое описание (короткое описание проблемы).

• Проект (название текущего проекта).

• Компонент приложения (в котором возник дефект).

• Версия (версия билда, в котором найден баг).

• Серьезность (градация степени влияния на приложение бага).

• Приоритет (очередь исправления бага).

• Статус (отображает статус бага в своем жизненном цикле).

• Автор (автор баг репорта).

• Назначение (кто должен исправить дефект).

**Окружение.**

• Операционная система, разрядность, Сервис Пак, браузер, его версия и т.д.

**Описание.**

• Шаги воспроизведения (описание пути, который приводит к возникновению дефекта).

• Фактический результат (результат, к которому приходим выполнив все шаги воспроизведения).

• Ожидаемый результат (результат, который быть в соответствии с требованиями).

**Дополнения.**

• Прикрепленный файл (логи, скриншоты, другие документы, которые могут помочь воспроизвести проблему или решить ее).

Несмотря на такое большое количество пунктов баг репорта, можно выделить несколько основных полей, присутствие которых необходимо:

• **Краткое описание.** Поле, в котором нужно поместить весь смысл всего баг репорта. Чаще всего, в коротком описании лаконично отвечают на 3 вопроса: «Где?», «Что?», «Когда?» (именно в такой последовательности, как бы не хотелось изменить ее по примеру всем известной игры).

• **Серьезность**. Дефект либо полностью останавливает работоспособность приложения, либо только часть функциональности, либо иное.

• **Шаги к воспроизведению.** Точное и понятное описание всех шагов, которые приводят к появлению дефекта, с учетом всех необходимых входных данных и т.д.

• **Фактический результат.**

• **Ожидаемый результат.**

**ПОДРОБНО**

**Баг-репорт** – это технический документ, описывающий ситуацию или последовательность действий приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Перечислим и подробно разберем основные атрибуты баг-репорта.

**Summary (Тема)**

Первый атрибут, который заполняется тестировщиком – это краткое описание бага (Summary).

В этом поле необходимо кратко изложить суть бага. Для того, чтобы в полном мере описать баг, необходимо последовательно ответить на три вопроса: ЧТО не так работает? в каком месте продукта, то есть ГДЕ? и КОГДА данная проблема проявляется?

**Например:**

Обязательные поля для заполнения не отмечены на форме регистрации после нажатия на кнопку «Отправить».

**Description (Подробное описание)**

В баг-трекере Mantis данное поле используется для более широкого описания сути бага, если есть такая необходимость. Во всех остальных популярных баг-трекерах (Jira, Redmine) поле Description используется для описания Шагов для воспроизведения и фактического/ожидаемого результатов.

В баг-трекере Mantis данные атрибуты (шаги и результаты) описываются в поле «Steps to reproduce».

**Steps To Reproduce (Шаги для воспроизведения)**

Необходимо точно описать все шаги, для того, чтобы можно было без особых проблем воспроизвести проблему.

**Actual/Expected Result (Результаты)**

**Actual result (Фактический результат)**–указывается что не так работает, в каком месте продукта и при каких условиях (то есть, описывая фактический результат необходимо опять-таки ответить на три вопроса Что? Где? Когда?). По сути, как правило, фактический результат должен быть аналогичен Summary.

**Expected result (Ожидаемый результат)** – указывается, как именно должна работать система по мнению тестировщика, например:

Обязательные поля для заполнения отмечены на форме регистрации после нажатия на кнопку «Отправить».

**Attachments (Вложения)**

К баг-репортам нужно прикреплять файлы, например, скриншот, видео или лог-файл. Это делается потому, что информация лучше усваивается визуально, чем текстово. Если добавлять скриншот, где визуально показать стрелочкой, что «вот тут вот баг», то, открыв этот скриншот иногда можно и не читать шаги  и description.

**Priority (Приоритет дефекта)**

Это атрибут, указывающий на очередность выполнения задачи или устранения дефекта. Можно сказать, что это инструмент менеджера по планированию работ. Чем выше приоритет, тем быстрее нужно исправить дефект.

**Один из вариантов градации Приоритета дефекта:**

*P1 Высокий (High)*  
Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта.  
*P2 Средний (Medium)*   
Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.  
*P3 Низкий (Low)*   
Ошибка должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.  
Порядок исправления ошибок по их приоритетам: High -> Medium -> Low

**Severity (Серьезность дефекта)**

Это атрибут, характеризующий влияние дефекта на работоспособность приложения.

**Один из вариантов градации Серьезности дефекта:**

*S1 Блокирующая (Blocker)*

Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.

*S2 Критическая (Critical)*

Критическая ошибка, неправильно работающая ключевая бизнес логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.

*S3 Значительная (Major)*

Значительная ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.

*S4 Незначительная (Minor)*

Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.

*S5 Неудобство (Tweak)*

Неудобство, очевидная проблема пользовательского интерфейса. Означает, что нужна «подгонка», повышение степени дружественности интерфейса.

*S6 Текст/опечатка (Text)*

Небольшая текстовая ошибка/опечатка. Пунктуационная или орфографическая ошибка.

*S7 Тривиальная (Trivial)*

Не касается бизнес логики приложения, плохо воспроизводимая проблема, малозаметная посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

**Status (Статус)**

Основной атрибут, определяющий текущее состояние бага. Отражает жизненный цикл бага от начального состояния до завершения.  
Названия статусов дефектов могут быть разными в разных баг-трекинговых системах, а также отличаться от [используемых на курсе](http://training.qatestlab.com/front-page/blog/course-materials/life-cycle-bug/).

**Стандартный набор статусов (для баг-трекера Mantis):**

*Новый (New)* – новый баг-репорт;  
*Обратная связь (Feedback)* – требуется дополнительная информация;  
*Рассмотрен (Acknowledged)* – с вопросом ознакомились, но подтверждения (дефекта) еще не было, ответственный не назначен;  
*Подтвержден (Accepted)* – дефект был воспроизведен и подтвержден;  
*Назначен (Assigned) –* баг-репорт назначен разработчику;  
*Отработан (Resolved)* – вопрос решен, ожидается подтверждение, что баг исправлен;  
*Закрыт (Closed)* – дефект исправлен, подтвержден и не требует никаких дальнейших действий.

Также следует указать **второстепенные атрибуты баг-репорта:**

**Component (Компонент) или Environment (Среда)** – это атрибут дефекта, который указывает на какой платформе этот дефект воспроизводится (iOS, Android, Windows, Mac и тд)

**Fix Version (Версия)** – этот атрибут указывает на каком этапе разработки программного продукта был обнаружен дефект

**Assignee (Назначение)**– это атрибут, в котором указывается личность, на которую данный баг-репорт назначается для дальнейшей проверки или исправления.

**Build (Номер сборки)** – это атрибут, в котором указан номер билда, в котором был обнаружен дефект.

**Lable (Лейбл)** – это атрибут, указывающий на принадлежность дефекта к тому или другому типу дефектов (графический дефект, дефект локализации и тд.)

32) Стохастичні критерії вибору тестів.

С.К. используют в случии стохастического тестирования. Стохастическое тестирование реализовано:

1. Разработать программу имитатор случайной последовательности, входных сигналов тестированного программного продукта.
2. Вычислить, независимым способом, значение выходных данных У кот соотв-ют Х. Т.о. мы получим тестовые наборы (Х, У).
3. Протестировать предложеные наборы.

Для контроля результата (процедуры Оракула) используют 2 метода:

1. Детерминированный контроль – простая проверка соотв полученого результата с тестовым набором.
2. Стохастический контроль – закон распределения выходных данных и закон распределения тестового набора совпадают.

Следовательно -> **сущ 2 критерия** на осн этой процедуры:

1. Cтатистический метод окончания тестирования

В осн лежит предположение о соблюдении гипотез закона распределения генерируемых данных и тест.набора.

1. Метод оценки скорости выявления ошибки. Этот критерий оценивает интервал времени между выявлениям текущей ошибки и последующей.

Формула вычисления время затрачиваемое на обнаружение N-ошибок:

Если tn+1> Td допустимого времени тестирования проекта, то тестирование заканчиваем, в противном случае продолжаем поиск ошибок.

**33) На прикладі системи MANTIS дайте характеристики системі відстеження помилок.**

**MantisBT** — [свободно распространяемая](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) [система отслеживания ошибок](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BE%D1%82%D1%81%D0%BB%D0%B5%D0%B6%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BE%D1%88%D0%B8%D0%B1%D0%BE%D0%BA) в программных продуктах (bugtracker). Обеспечивает взаимодействие разработчиков с пользователями ([тестировщиками](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D1%89%D0%B8%D0%BA)). Позволяет пользователям заводить сообщения об ошибках и отслеживать дальнейший процесс работы над ними со стороны разработчиков.

Система имеет гибкие возможности конфигурирования, что позволяет настраивать её не только для работы над программными продуктами, но и в качестве системы учёта заявок для [Helpdesk](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B1%D0%B0_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B9_%D0%BF%D0%BE%D0%B4%D0%B4%D0%B5%D1%80%D0%B6%D0%BA%D0%B8).

Возможна интеграция с wiki-движком для создания документации ([DokuWiki](https://ru.wikipedia.org/wiki/DokuWiki)).

Система является веб-приложением, поэтому не требует для работы специального ПО на стороне клиента и работает через [веб-браузер](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80).

Название Mantis ([богомол](https://ru.wikipedia.org/wiki/%D0%91%D0%BE%D0%B3%D0%BE%D0%BC%D0%BE%D0%BB)) происходит от того, что богомол питается жуками (bug).

Плюсы:

* Бесплатность и свободность, лицензия GNU General Public License (GPL)
* Код на PHP свободно модифицируем
* Понятно написанный код
* Цветовая индикация по статусу инцидента (бага)
* Настраиваемые пользователем поля
* Удобные фильтры
* Скорость работы
* Уведомления по e-mail
* Большое количество плагинов, расширяющих функциональность

Минусы:

Через веб-интерфейс нельзя произвести существенные изменения настроек. Необходимо настраивать в конфигурации. Через интерфейс можно редактировать возможность перехода между статусами, но не список статусов. Изменить (добавить, удалить) имеющиеся поля в фильтре, окнах создания и просмотра бага можно только редактируя код. Но данные операции с кодом достаточно просты и не требуют глубоких знаний программирования на PHP.

**34) Тестирование таблицы переходов состояний.**

Тестирование таблицы переходов — это метод тестирования «черного ящика», который используется там, где некоторый аспект системы может быть описан в так называемом «конечном автомате». Это просто означает, что система может находиться в (конечном) числе разных состояний, а переходы из одного состояния в другое определяются правилами «машины».

Это модель, на которой базируются система и тесты. Любая система, в которой вы получаете другие выходные данные для одного и того же ввода, является системой с конечным состоянием.

Кстати, наши курсы Тестирования ПО в Минске тщательно разбирают этот вопрос. Примеры тестирования таблицы переходов.

Если вы попросите снять 100 фунтов стерлингов с банковского банкомата, вам могут

быть выданы наличные. Позже вы можете сделать точно такой же запрос, но вам будет отказано в деньгах (потому что вашем балансе недостаточно средств).

Этот более поздний отказ объясняется тем, что состояние вашего банковского счета изменилось с достаточного количества средств для их вывода к нехватке средств. Транзакция, которая привела к изменению вашего состояния вашего счета, была, вероятно, более ранним выходом.

Другим примером является текстовый процессор. Если документ открыт, вы можете его закрыть. Если ни один документ не открыт, то «Закрыть» недоступен. После того, как вы выберите «Закрыть» один раз, вы не можете выбрать его снова для того же документа, если вы не откроете этот документ. Таким образом, документ имеет два состояния: открытое и закрытое.

Модель перехода состояний состоит из четырех основных частей:

* Состояния, которые может принимать программное обеспечение (открытые / закрытые или финансируемые / недостаточные фонды);
* Переходы из одного состояния в другое (разрешены не все переходы); Действия, которые вызывают переход (вывод денег, закрытие файла);
* Действия, которые возникают в результате перехода (сообщение об ошибке или получение денежных средств).

Обратите внимание, что переход не должен изменяться. Он может оставаться в том же состоянии. Фактически, попытка ввода недопустимого ввода, скорее всего, приведет к появлению сообщения об ошибке в качестве действия, но переход будет в том же состоянии, в котором система была до этого.

Получение тестовых сценариев из модели перехода между состояниями является методом «черного ящика». Процесс измерения того, насколько вы протестировали (охватили), близок к процессу тестирования белого ящика. Тем не менее, тестирование переходного состояния обычно рассматривается как метод черного ящика.

Вы можете проектировать тесты для проверки каждого перехода, показанного в модели. Если проверяется каждый (действительный) переход, это называется покрытием «0-переключателя».

Вы также можете проверить ряд переходов через более чем одно состояние. Если вы охватили все пары из двух допустимых переходов, вы бы имели покрытие

«1-переключатель», охватывая наборы из 3-х переходов, обеспечив покрытие

«2-коммутатора» и т. Д.

Однако вывод тестов только из модели может опустить отрицательные тесты, в которых мы могли бы попытаться генерировать недопустимые переходы. Чтобы увидеть общее количество комбинаций состояний и переходов, действительных и недействительных, можно использовать таблицу состояний

**35) Ключові засади автоматизації тестування**

Во-первых, АТ экономит время. Программа-робот гораздо быстрее "щелкает по кнопкам" или посылает запросы по протоколу, чем любой человек. Значит, программисты быстрее узнают об ошибках и исправят их. И компания получит преимущество перед конкурентами.

Во-вторых, она исключает человеческий фактор. Когда вы целый день выполняете одни и те же или очень похожие действия, то вероятность ошибок к вечеру резко возрастает. Поручите рутинные операции роботу, он не ошибется, а сами займитесь чем-то более творческим, например планированием.

В-третьих, АТ дает возможность работать без графического пользовательского интерфейса. Часто бывает так, что на ранних этапах развития программного продукта этот интерфейс еще не согласован и ПО представляет собой по сути набор каких-то модулей и элементов, которые тем не менее должны правильно работать. Или, например, на одной из итераций добавляют функцию в почти готовый продукт, так что доступа через интерфейс к ней нет. Что уж говорить об обмене данными по протоколам, который может происходить вообще незаметно для пользователя. Хорошие средства АТ могут существенно помочь в таких случаях.

В-четвертых, средства автоматизации позволяют эмулировать многопользовательскую работу для нагрузочного тестирования. Если ваша программа не поддерживает многопользовательский режим, то ее можно запустить на несколько суток и посмотреть, что по прошествии их будет и с программой, и с компьютером. Если же поддерживает и рабочей нормой считается одновременное обращение к вашему приложению нескольких тысяч пользователей, то средства АТ являются единственным способом решить проблему нагрузочного тестирования. Конечно, можно попробовать посадить за компьютеры весь тестовый отдел или даже всю фирму, но согласитесь, что это будет крайне трудоемко. Средства же для нагрузочного АТ позволяют гибко эмулировать многопользовательскую работу так, как вам нужно.

Наконец, практически все средства АТ имеют инструментарий фиксации ошибок и результатов. Они позволяют моделировать различные ошибочные ситуации, строить любые отчеты и диаграммы по вашему вкусу. Иногда это бывает очень удобно, но все же увлекаться этим не стоит, так как результаты вашей деятельности будет оценивать человек, которому обычно нужен минимум четкой и ясной информации.

В нефункциональных требованиях к приложению должно быть явно выделено и описано требование по тестируемости, с тем, чтобы выбираемые технологии, платформы и инструменты позволяли выполнять автоматизацию тестирования.

Архитектура приложения должна быть модульной или разделенной по слоям, с тем, чтобы каждый слой мог быть протестирован независимо от остальных. Например, распространенная архитектура MVC (model view controller) - хорошо тестируема на каждом уровне и вы можете применить все современные практики по автоматизации тестирования.

Тестами должны быть покрыты практически все участки вашего приложения. В этом вам поможет методика test driven development (TDD). Суть методики заключается в том, что тесты (модульные или приемочные) создаются перед тем, как реализуется логика приложения. Таким образом, заранее продумывается контракт, интерфейсы со стороны внешнего пользователя (например, другой программы или класса)

36) Особливості інтеграційного тестування для об’єктно-орієнтовного програмування.

В соотв с ОО парадигмой карта вызовов будет существенно отличатся, это фиксированая диаграмма последовательности. Она строится после диаграммы классов, где вертикальные ячейки отображают сами классы, переходы означают вызовы методов различных классов.

Длина вертикальной линии – время выполнения метода.

В современом программном окружении ООП явл событийно управляемым. Передача управления внутри программы(объекта) выполняется не только через активный вызов, а еще через генерацию сообщений другим объектам.

Другая особеность – как таковое програмирование заменяется стандартом Look&Feel ( вызуальным связыванием объектов).

Для того что бы выполнить работу по тестированию необходимо выявить 2 – способа связи между объектами в тестир.иерархии.

1. Прямой вызов(на прямую вызыват 1-метод из другова – Р путь)
2. Обработка сообщений (ММ – путь)

Р-путь возникает в случии вызова командной строки, кот обрабатывает алгоритм и выгружается из памяти. При этом оно может создать ММ-путь.

37) Структурні критерії вибору тестів.

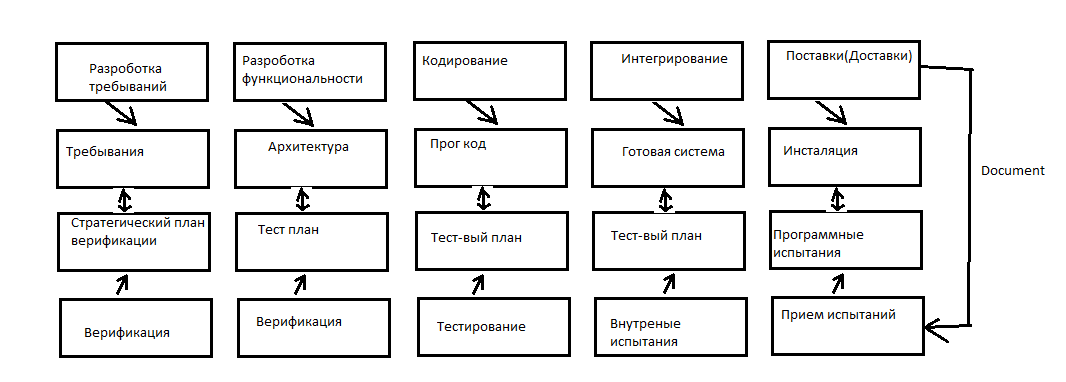
Базируется на идеи белого ящика (программный код доступен для тестировщика).

Сущ 3 осн критерия:

1. С0 – тестирование команд – это означает что необходимо создать множ.тестовых наборов. При выполнении кот каждый оператор срабатывает хотя бы 1 раз.
2. С1 – тестирование ветвей – создать набор тестов кот обеспечивает прохождения каждой ветки алгоритма минимум 1 раз.
3. С2 – тестирование путей – создать набор тестов обеспечивающих прохождения путей минимум 1 раз.

38) Документування в процесі верифікації.

Жизненный цикл



39) = 21)

40) Визначення якості программного забезпечення (ISO, IEEE).

Качество ПО (по ISO) – 1) это полнота свойств и характеристик продукта, процеса, услуги, кот обеспечивает способность удовлетворять заявленным или подразумевающимся потребностям.

Качество ПО (по **IEEE)**– 2) это степень в которой ПО обладает требуемой комбинацией свойств ISO (factor, criteria, metrics, evaluated element).

Визначення цих характеристик і атрибутів за стандартом ISO 9126:2001:

1. Функціональність (functionality). Здатність ПЗ в певних умовах вирішувати задачі, потрібні користувачам. Визначає, що саме робить ПЗ, які задачі воно вирішує.
   1. suitability. Совместимость в смысле алгоритма.
   2. Точність (accuracy). Соответсвие между ф-ей и требываниями.
   3. Здатність до взаємодії (interoperability). Здатність взаємодіяти з потрібним набором інших систем.
   4. Відповідність стандартам і правилам (compliance).
   5. Захищеність (security).
2. Надійність (reliability).
   1. Зрелость надожности (maturity).
   2. Стійкість до відмов (fault tolerance).
   3. Здатність до відновлення (recoverability).
3. Зручність використання (usability) або практичність.
   1. Зрозумілість (understandability).
   2. Зручність навчання (learnability).
   3. Зручність роботи (operability).
   4. Привабливість (attractiveness).
4. Продуктивність (efficiency) або ефективність.
   1. Поведение во времени (time behaviour).
   2. Ефективність використання ресурсів (resource utilisation).
5. Зручність супроводу (maintainability).
   1. Аналізованість (analyzability) або зручність проведення аналізу.
   2. Зручність внесення змін (changeability).
   3. Стабільність (stability).
   4. Зручність перевірки (testability).
6. Переносимість (portability).
   1. Адаптованість (adaptability).
   2. Зручність установки (installability).
   3. Здатність до співіснування (coexistence).
   4. Зручність заміни (replaceability) іншого ПЗ даним.

1,2,3 – Разработчик

4,5,6 - Пользователь