В последние несколько лет **тестирование ПО** в отечественных компаниях наконец-то превратилось не просто в поиск ошибок или же средство получения информации о состоянии проекта, а в мощный инструмент разработки, позволяющий достичь нужного уровня качества.

Тестирование – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением на ограниченном наборе тестов, выбранном определенным образом.

Testing is a process of executing a program with intend of finding errors.

Test is the act of executing software with test cases with an objective of:

* Finding failure;
* Demonstrate correct execution.

Тестировщик:

1. Управляет выполнением программы, создает искусственные ситуации
2. Наблюдает за поведением программы и сравнивает наблюдаемое поведение с ожидаемым.

**Testing Axioms**

#### It's Impossible to Test a Program Completely

Пример: даже такое просто приложение как калькулятор слишком тяжело протестировать полностью. Представьте себе, что вам необходимо протестировать калькулятор. Вы решили начать со сложения. Вы пытаетесь сложить 1+0=. Вы получаете ответ 1. Это правильно. Затем пытаетесь сложить 1+1=. Вы получаете 2. И что дальше?? Калькулятор способен складывать 32-разрядные числа, таким образом, что вы должна попробовать все возможные варианты. А если вы начнете тестировать десятичные числа. И т. д. А когда вы проверите, что обычные числа складываются правильно, вы должны попробовать неправильные вводы, чтобы убедиться, что они работают правильно. Помните, что вы не ограничены нажатием клавиш и с клавиатуры. Вы также можете попробовать протестировать буквенные значения: 1+а, z+1, 1f1+2b2, …

Калькулятор позволяет нажатие клавиш Backspace и Delete, поэтому вы должны протестировать и их. 1<backspace>2+2 должно равняться 4. Все, что бы вы не ввели должно быть протестировать совместно с нажатием Backspace для одного или нескольких вводов. И затем вы можете протестировать сложение 3-х, 4-х чисел и т. д. И это только для сложения. А есть еще вычитание, умножение. Деление, проценты.

Таким образом, мы увидели, что невозможно полностью протестировать программу, даже такую простую как калькулятор.

#### Software Testing is a Risk-Based Exercise

Если вы решили не тестировать программу полностью, то вы в некоторой степени рискуете. Вот в примере с калькулятором. Если вы не хотите тестировать сценарий 1024+1024=2048, то ведь есть вероятность, что программист допустил ошибку в сложении именно этих 2-х чисел. Вы это не протестировали, а клиент это заметил, и он найдет эту ошибку. И это будет дорогой баг, так как его найдет именно клиент.

Так как вы все протестировать не сможете, то, вероятно, вы пропустите некоторые баги. Поэтому ключевой концепцией является то, что тестировщикам необходимо изучить, как избежать выполнения огромного количества тестов и как определять: что важно протестировать, а что нет.

На графике показана зависимость между количеством тестов и количеством найденных багов. Если вы попытаетесь протестировать все, то стоимость тестирования возрастет колоссально и количество пропущенных багов уменьшается до точки, когда стоимость такого тестирования становится неоправданной. Если вы в некоторой точке прекратите тестирование, то стоимость тестирования будет маленькой, но вы пропустите большое количество багов. Поэтому целью будет найти эту оптимальную точку, когда количество тестов будет ни слишком большим, ни слишком маленьким.

#### Testing Can't Show That Bugs Don't Exist

Представьте, что есть дезинсектор, который осматривает дом на наличие насекомых. Он видит живых багов, мертвых или личинок – тогда он с уверенностью может сказать, что баги в доме есть.

Он посещает другой дом. В этот раз он не обнаруживает никаких следов багов. Он заглядывает во всевозможные места и не находит очагов заражения. Возможно, он найдет пару мертвых жуков или старых, но живых и здоровых он не обнаруживает. Можно ли с уверенностью сказать, что в доме больше нет насекомых? Нет. Можно только сделать вывод, что он просто не нашел живых багов. Если конечно не просмотреть весть дом с фундамента и до крыши, но сказать с уверенностью, что вы не пропустили ни одного, нельзя.

В тестинге все точно так же, как с дезинсектором. Можно показать, что баги есть, но утверждать, что их нет - нельзя. Вы можете выполнить тестовые кейсы, найти и зарегистрировать баги, но нет гарантии, что еще какие-нибудь баги не обнаружатся. Вы можете продолжить тестинг и, возможно, найти еще что-то.

**Важной причиной тестирования является предотвращение дефектов. Вы можете проходить тесты, находить и заводить дефекты, но гарантировать, что багов нет, вы не можете.**

#### The More Bugs You Find, the More Bugs There Are

Баги, найденные в приложении, можно сравнивать с настоящими багами. А все почему? Если вы находите ошибку, то очень большая вероятность того, что другие ошибки вы найдете где-то рядом. Почему так происходит?

- У программистов тоже бывают плохие дни. Как у всех людей. Код, написанный в один их удачных дней, может быть превосходным, но в другой день может быть плохим. Ошибка может свидетельствовать про то, что другие проблемы могут быть где-то близко.

- Программисты часто допускают одни и те же ошибки. У каждого человека есть свои привычки. Программист, который допустил однажды ошибку, с большой вероятностью допустит её еще раз.

- Некоторые баги действительно могут быть «the tip of iceberg». Часто дизайн и архитектура приложения имеют фундаментальные ошибки. Тестер, который найдет несколько багов и которые на первый взгляд могут показаться несвязанными, в действительности могут иметь общую причину.

**You can’t guarantee quality.**

Как тестировщик, вы не можете протестировать все и не отвечаете за качество продукта. Важно, что у тестировщика нет возможностей гарантировать качество, т.к. задача тестировщика – искать и находить проблемы.

**Be a customer.** Быть клиентом. Попробуйте использовать систему в качестве пользователей. Чтобы понять это, поручите человеку, который не имеет представления о приложении, попользовать его некоторое время, и вы будете поражены, увидев количество проблем, с которыми этот человек столкнется. Проводя тестирование таким образом, можно столкнуться со множеством неожиданных испытаний - стрессы, нагрузки и т.д.

What makes a good tester?

* **Know the technology**. Знание технологии, с помощью которой разрабатывается приложение. Это является существенным преимуществом любого тестера. Это помогает лучше проектировать дизайн, зная слабые или сильные стороны этой технологии. Хорошие тестеры знают, что приложение будет поддерживать, а что нет. И таким образом,   
  сосредоточившись на этих основных пунктах, в дальнейшем это поможет им быстро найти дефекты в приложении.
* **Perfectionist and a realist.** Будучи перфекционистом, это поможет тестерам обнаружить проблемы и будучи реалистом - помогает знать и выделить действительно важные проблемы. Вы будете знать, какие из них требуют исправления в первую очередь, а какие нет.
* **An explorer**. Немного творчества и отношение к риску помогает тестерам в поиске неизвестных ранее ситуаций и находить такие ошибки, которые в противном случае не заметили бы.
* **Troubleshooting**. Устранение неполадок и выяснение, почему что-то не работает, помогает тестерам быть уверенными в действительности найденной проблемы и ясно излагать дефект разработчикам.
* **Possess people skills and tenacity.** Тестеры могут столкнуться сильным сопротивлением программистов. Быть умным и социально-дипломатичным не значит, что нужно быть нерешительным. Лучшие тестеры одновременно и неконфликтны-коммуникабельны, и упорные, где это имеет значение.
* **Organized.**  Лучший тестер очень хорошо понимает, что он тоже может ошибаться, но попытка для тестирования всего одна. Он очень хорошо организован и имеют checklists для проверки, использует файлы, факты и цифры для подтверждения своих выводов, которые могут быть использованы в качестве доказательств и подтверждения своих выводов.
* **Objective and accurate.** Объективные и точные. Неточная информация теряет очень много доверия. Хорошие тестеры должны убедиться, что их выводы точные и воспроизводимые.
* **Дефекты ценны.** Хорошие тестеры учатся на дефектах. Каждый дефект - это возможность узнать новое и совершенствоваться дальше. Дефект, обнаруженный на ранней стадии, существенно дешевле по сравнению с найденным в более позднем этапе. Дефекты могут вызвать серьезные проблемы, если их не удалось найти вовремя. Изучение дефектов помогает предотвратить проблемы в будущем, отслеживать улучшения продукта, улучшать прогнозирование и оценки возникновения проблем в будущем.

Application Inputs and Outputs

Программа представляет собой механизм по переработке информации. На вход поступает информация в каком-то одном виде, на выходе – в другом. При этом входов и выходов у программы может быть много, они могут быть разными, т.е. у программы могут быть разные интерфейсы для общения с внешним миром, и эти интерфейсы могут иметь разные виды. Наиболее распространенные интерфейсы это:

- пользовательский интерфейс (UI); - графический или текстовый консольный, или речевой. Через этот интерфейс компьютер взаимодействует с человеком.

- программый интерфейс. (API) Через этот инерфейс программы взаимодействуют друг с другом, а не с человеком

- сетевой протокол – для взамодействия программ друг с другом, но через сеть, а не непосредственно;

- файловая система – программы могут писать данные на диск и читать данные с диска.

- состояние окружения - которое могут програмы модифицировать.

- события, например таймер.

Тестировщик создает искусственные ситуации и смотрит как программа себя ведет.

Схема тестирования примерно такова. Тестировщик на вход получает программу и требования, после этого он с ними что-то делает, смотрит на поведение программы в опеределенных ситуациях и получает информацию о несоответствиях. Эта информация далее используется для того, чтобы улучшить программу либо изменить требования, т. е. меняться может и то и другое.

Software Architecture

**Клиент-сервер (**[**англ.**](http://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)**Client-server)** — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг (сервисов), называемыми **серверами**, и заказчиками услуг, называемыми **клиентами**.

Бывают ситуации, когда и клиент, и сервер располагаются на одной машине, тогда такой сервер называют **локальным**.

Модель клиент-серверной орхитектуры определяется распределением обязаннстей между клиентом и сервером. Есть 3 уровня операций:

- уровень представления данных – это по сути интерфейс пользователя и отвечает за представление данных пользователя и ввод от него команд;

- прикладной уровень – реализует основную логику использования, на котором осуществляется обработка информации;

- уровень управления данными, который обеспечивает хранение данных и доступ к ним.

**Двухуровневая** архитектура предполагает взаимодействие 2-х программных модулей – клиентского и серверного. В зависимоти от того, как между ними распределяются функции, различают:

- модель тонкого клиента – вся логика использования и управления данными сосредоточена на сервере. Клиентская программа обеспечивает функции только уровня представления;

- модель толстого клиента – сервер только управляет данными, а обработка информации и интерфейс сосредоточены на стороне клиента.

**Трёхслойная архитектура.** предполагает наличие следующих компонентов приложения:

* клиентское приложение (обычно говорят «[тонкий клиент](http://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BD%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BB%D0%B8%D0%B5%D0%BD%D1%82)» или терминал),
* сервер приложений,
* сервер [базы данных](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).

**Пример 3-х звенной архитектуры клиент-сервер**

Компоненты трёхзвенной архитектуры, с точки зрения программного обеспечения реализуют определенные сервера БД, web-сервера и браузеры. Место любого из этих компонентов может занять программное обеспечение любого производителя. Ниже представлено описание взаимодействия компонентов трехуровневой архитектуры клиент-серверного приложения.

1. Сервер БД представлен Oracle-сервером;
2. сервер приложений технологиями: ADO.NET, ASP.NET и web-сервером IIS;
3. роль клиента выполняет любой web-браузер.

**Браузер клиента** *1->* **Сервер IIS** *2->* **Исполняющая среда ASP.NET 2.0** *3->* **Провайдер данных ADO.NET 2.0** *4->* **Сервер MySQL** *5->* **Провайдер данных ADO.NET 2.0** *6->* **Исполняющая среда ASP.NET 2.0** *7->* **Сервер IIS** *8->* Браузер клиента

* 1 — браузер клиента отправляет HTTP-запрос;
* 2 — на стороне сервера служба Web [Internet Information Services](http://ru.wikipedia.org/wiki/Internet_Information_Services) (web-сервер IIS) определяет тип запрашиваемого ресурса, и для случая запроса \*.aspx (расширение файлов страниц ASP.NET) загружает соответствующее ему (запросу) расширение Internet Server Application Programming Interface (ISAPI). Для страниц aspx это расширение isapi\_aspnet.dll. IIS также осуществляет идентификацию и авторизацию пользователя от которого поступил запрос. В свою очередь расширение isapi\_aspnet.dll загружает фабрику обработчиков ASP.NET. Далее, фабрика обработчиков создает объектную модель запрашиваемой страницы и обрабатывает действия пользователя.
* 3 — в ходе генерации ответа приложению ASP.NET может потребоваться обращение к БД, в этом случае используя библиотеки классов провайдера данных ADO.NET 2.0, выполняющая среда обращается к серверу БД;
* 4 — провайдер данных ADO.NET 2.0 передает запрос на операцию с БД серверу MySQL;
* 5 — сервер MySQL осуществляет обработку запроса, выполняя соответствующие операции с БД ;
* 6 — провайдер данных ADO.NET 2.0 передает результаты запроса объекту страницы;
* 7 — объект страницы с учетом полученных данных осуществляет рендеринг графического интерфейса страницы и направляет результаты в выходной поток;
* 8 — сервер IIS отправляет содержимое сгенерированной страницы клиентскому браузеру.

**Приемущества:**

* Делает возможным распределение функций вычислительной системы между несколькими независимыми компьютерами в сети. Это позволяет упростить обслуживание вычислительной системы. В частности, замена, ремонт, модернизация или перемещение сервера не затрагивают клиентов.
* На сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа. Все данные хранятся на сервере, который защищён гораздо лучше большинства клиентов.
* Позволяет объединить различные клиенты. Использовать ресурсы одного сервера часто могут клиенты с разными аппаратными платформами, операционными системами и т. п.

**Недостатки:**

* Неработоспособность сервера может сделать неработоспособной всю вычислительную сеть.
* Поддержка работы данной системы требует отдельного специалиста — системного администратора.
* Высокая стоимость оборудования.

## Software versioning

**Software versioning** — это процесс создания уникальных имен или номеров для различных версий продуктов программного обеспечения.

При имеющейся категории номера версии (главная, второстепенная), номера обычно выставляются в возрастающем порядке и соответствуют новым разработкам в программном обеспечении. На начальном уровне отслеживанием постепенно появляющихся версий электронной информации занимается система управления версиями, позволяющая хранить несколько версий одного и того же документа, при необходимости, возвращаться к более ранним версиям, определяя, кто и когда сделал то или иное изменение и многое другое. Вместе с тем для отслеживания изменений программного обеспечения было создано большое количество схем присвоения номеров версиям.

Каждой стадии разработки программного обеспечения соответствует уникальный идентификатор, который состоит из последовательности номеров или букв. В некоторых схемах последовательные идентификаторы используются для определения значимости перемен между стадиями разработки: эти перемены классифицируются по уровням значимости. Решение о том какую последовательность изменить между стадиями разработки основано на значимости перемен на последней стадии разработки, например в схеме, с версией, состоящей из последовательности 4 чисел, первое число может быть прибавлено только тогда когда код полностью переписан, в то время как четвертое число изменяется при незначительных переменах в интерфейсе или документации.

В более поздних релизах, главное число (major) увеличивается, когда происходят значительные переходы в функциональности, второстепенное число (minor) прибавляется только тогда, когда были добавлены незначительные функции или внесены исправления. Номер версии изменяется, если исправлены все мелкие неполадки. Для типичного программного продукта используются следующие номера: *0.9 (для бета-версии), 0.9.1, 0.9.2, 0.9.3, 1.0, 1.0.1, 1.0.2, 1.1, 1.1.1, 2.0, 2.0.1, 2.0.2, 2.1, 2.1.1, 2.1.2, 2.2*, и т.д. Разработчики порой перескакивают от версии 5.0 сразу к 5.5, для того чтобы обозначить добавление нескольких значимых функций в программе, однако их не достаточно, чтобы изменить главный номер версии, тем не менее такие скачки все же неуместны.

### Обозначение стадии разработки

   Для присвоения альфа или бета статуса релизам, которые еще не достаточно стабильны для практического применения, существуют схемы, где в первой последовательности используется ноль. Он ставится третьим числом в тех случаях, когда планируется еще тестировать версию или версия годна только для внутреннего применения.  
  
Последовательность в таких схемах следующая:

* **0** — альфа
* **1** — бета
* **2** — релиз кандидат
* **3** — публичный релиз

Например:

* **1.2.0.1** вместо **1.2-a**
* **1.2.1.2** вместо **1.2-b2** (бета с несколькими исправленными ошибками)
* **1.2.2.3** вместо **1.2-rc** (релиз кандидат)
* **1.2.3.0** вместо **1.2-r** (для коммерческого распространения)
* **1.2.3.5** вместо **1.2-r5** (для коммерческого распространения со многими исправленными ошибками)

### Приращение последовательности

   Существует два разных способа приращения последовательности номеров в версии. Большинство продуктов свободного программного обеспечения используют непрекращаемый поток последовательных номеров: *1.7.0*, *1.8.0*, *1.8.1*, *1.9.0*, *1.10.0*, *1.11.0*, *1.11.1*, *1.11.2*, и т.д. Примером такого продукта может служить [MediaWiki](http://en.wikipedia.org/wiki/MediaWiki). В других программах используются десятичные номера: *1.7*, *1.8*, *1.81*, *1.82*, *1.9*, и т.д. В таких программах после версии *1.8* будет идти версия *1.81*, текущие релизы будут обозначаться *1.81a*,*1.81b*, и т.д.

### Значимость нумерации версий в разработке программного обеспечения

   Номера версий используются в практических условиях потребителем или клиентом для того, чтобы можно было сравнить имеющуюся у них копию продукта программного обеспечения и новую версию, выпущенную разработчиком. Команда программистов и компании используют нумерацию версий для сравнения отдельных частей и секторов программного кода одних версий с другими, обычно сотрудничая с Системой контроля версий. Не существует абсолютной и определенной схемы нумерации версий продуктов программного обеспечения, поэтому очень часто нумерация зависит от личного выбора программистов.

<http://habrahabr.ru/blogs/development_tools/118756/#habracut>

**Качество**

Стандарт ISO (International Organization for Standartization) 9126. Согласно этому стандарту можно выделить 6 аспектов качества, у которых можно выделить некоторое кол-во характеристик.

- функциональность;

- надежность (reliability);

- практичность;

- эффективность;

- сопровождаемость;

- переносимость.

Функциональность – основной аспект качества и наиболее важной его характеристикой является **пригодность к использованию (suitability)**. Т. е. программа должна делать то что она должна делать, для чего она предназначена.

Во-вторых она должна делать это правильно. Она должна вычислять все с определенной точностью, должна правильно конвертировать данные и т. д. **Точность, правильность (accuracy).**

Способность к взаимодействию с другими программами, с ОС, с другими версиями той же программы. **Способность к взаимодействию (interoperability)**

В частности она должна проддерживать какие-то стандарты и удовлетворять определенным правилам. **Соответствие стандартам и правилам (compliance).**

Интересной особенностью этого стандарта явл. **защищенность (security).** Этот стандарт не выделяется как аспект верхнего уровня, а считается подхарактеристикой функциональности. Пр-ма не должна разрушать данные, не должна мешать работеть другим программам, не должна предоставлять доступ к данным тем, кому этот доступ не разрешен.

Надежность:

**Зрелость (maturity)** которая является обратной величиной к частоте отказов.

**Устойчивость к отказам (fault tolerance)** – способность системы не реагировать на какие-то внутренние проблемы;

**Способнсоть к восстановлению работоспособности при отказах (recoverability)**. Если сервер упал, то система должна восстановиться, вернуть все необходимые данные, ничего не потерять и продолжить работу.

Практичность:

**Понятность (understandability)** пользователь должен понять ка воспользоваться программой для достижения своих целей;

**Удобство обучения (learnability)** – у прогарммы должна быть какая-то документация

**Работоспособность или управляемость (operability)** – пользователь должен иметь возможноть управлять поведением программы, она должна реагирвоать на его действия

Привлекательность (attractiveness) – эстетическая привлекательность.

Эффективность или производительность:

**Временные характеристики (time behaviour)** – время отклика, скорость работы программы, скорость обработки определенных данных и т д.

**Использование ресурсов (resource utilisation)** – использование дисковых ресурсов, использование процессора, оперативной памяти, сетевых ресурсов.

Сопровождаемость:

Этот аспект важен не сколько юзерам, сколько самим разработчикам, поэтому является внутренним аспектом, а не внешним.

**Анализируемость (analyzability)** -

**Изменяемость, удобство внесения изменений (changeability)**

риск возникновения неожиданных эффектов при внесении изменений **(stability)**

контролируемость или удобство тестирования программы **(testability)**

Переносимость или мобильность:

Пр-ма должна работать в различных окружениях, быть достаточно проста в установке, должна уменять работать с другими программами и не мешать им и она не должны мешать ей должна обеспечивать пользователь легко перейти от использования другого ПО на эту программу, т. е. предоставить какие-то возможности по миграции, чаще всего это миграции данных.

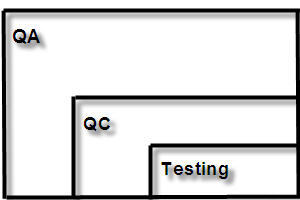
**Адаптируемость (adaptability)**

**Устанавливаемость, удобство установки (installability)**

**Способность к сосуществованию с другими ПО (coexistence)**

**Удобство замены другого ПО данным (replaceability)**

В соответствии с данными аспектами качества можно выделить аналогичные виде тестирования.



**Testing**(тестирование) - это самый низкий уровень - прохождение тест кейсов и локализация дефектов… В принципе на это способны люди и без специального образования…  
**QC**- следующий уровень - контроль качества продукта - анализ результатов тестирования и качества “билдов”, в процессе разработки.  
**QA**- решает более глобальные задачи. Анализируя работу тестировщиков и QC, в случае возникновения проблем, вовремя находит пути ее решения и не дает ей развиться и повлиять на качество продукта.

**QA** (**Quality Assurance**, **Обеспечение качества**) - совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации ПО информационных систем, предпринимаемых на разных стадиях **жизненного цикла ПО**, для **обеспечения качества выпускаемого продукта**.  
**QC** (**Quality Control**, **Контроль качества**) - совокупность действий проводимых над объектом тестирования в процессе разработки для получения информации об актуальном состоянии объекта тестирования в разрезах: "**готовность Продукта к выпуску**", "**Соответствие зафиксированным требованиям**", "**Соответствие заявленному уровню качества продукта**".  
**Software Testing** является одной из техник контроля качества и включает в себя активности по планированию работ (**Test Management**), проектированию тестов (**Test Design**), выполнению тестирования (**Test Execution**) и анализу полученных результатов (**Test Analysis**).

<http://alexeybulat.blogspot.com/2007/12/qa-qc-testing.html> - продолжение здесь

## PDS responsibilities

Product specialists and Business Analysts – persons, who communicate directly with clients, discuss and interpretate there needs to functional requirements that are base for creating issues and CRs for developers and QA.

PDS (product specialist) responsibilities includes strong understanding of product functionality and features to participate in discussions and making decisions on covering and fulfilling requirements and needs of a client. To recognize whether particular case is defect or specific feature, to achieve concurrency with clients on how specific piece of functionality should work.

## BA responsibilities

BA (Business analyst) responsibilities includes all product specialists responsibilities and additionally clear vision of product features to have ability to convince clients that current problem is some specific feature or it is result of not correct usage or setup, etc. Also convince client that all his needs are fulfilled in particular product and he doesn’t need anything else, writing functional requirements.

## Working with requirements

BA, after concurrency with client is achieved, is writing functional requirements that should be implemented in next versions of product by developers and covered by test cases and after that tested. Than BA reviews that functionality and cases. If everything is ok, he makes meeting with PDA and clients (usually it is a conference call) and presents changes to the client. Client makes decision whether this solution fulfills his goals, and approves or rejects it.

**Product Management:**

Reviews new functionalities of a product

Reviews requirements and functional specifications for testing

Reviews and signs Test Plan

Reviews and approves Test Cases and reviews Test Sets written by SCC and Contractor

Verifies criticality of defects

Performs Exploratory Testing and System Testing as appropriate

Recommends corrective actions and follow-up on testing

Performs final review and signs off on all system Testing Summary Reports

#### Testing Manager:

Creates and monitors QC testing project plan for contractors

Monitors all testing activities

Requests creation of the environments for testing and verifies the software is installed properly

Creates tasks for all testing activities (creating Test Cases, running Test Cases)

Creates Test Sets in SoftTest for testing activities

Reconciles Test Cases from Test Set with Change List to ensure every change in software is covered by Test Case and the test case is executed

Produces and verifies Testing Summary Report

#### Quality Control/Product Testers

Designs and creates Test Cases for Manual Functional, Integration and System Testing

Shadows Test Cases and execution with testers/contractors

Maintains testing environments

Performs Functional, System, and Installation Validation Testing across new and existing applications

Streamlines and enhances the testing lifecycle to ensure test planning, execution, and reporting is effective, efficient, standardized, coordinated, and integrated

Maintains all types of testing documentation

Responsible for the knowledge of their Product, including continuing education concerning changes and enhancements by using SCC software courses and training sessions

SCC maintains five levels of the quality system documentation:

         Level 1 - Quality Manual, Quality Policy and Organizational Charts (described in Section 4.1)

         Level 2 - Standard Operating Procedures (SOP)

         Level 3 - Guidelines and standards

         Level 4 – Work instructions, drawings, and specifications

         Level 5 – Quality Records – TMS Tasks, templates, forms, checklists, logs, spreadsheets, meeting minutes, and memos

**Level 1** – The Quality System Manual is the first tier document that describes the main responsibilities, functions, and activities that directly support the Quality System.  The Quality System Manual identifies the fundamental requirements for lower-level documents.  The Quality System Manual is structured to comply with the QSR.

**Level 2** – Procedures (Standard Operating Procedures) are second tier documents that provide a general outline of the methods employed in support of Quality System Manual elements and the responsibilities for carrying them out.  Standard Operating Procedures specify departments or functions responsible for carrying out the high-level steps of a process.

**Level 3** – Guidelines and Standards are third tier documents that provide instructions for performing daily tasks.

**Level 4**– Work instructions, drawings, and specifications are fourth tier documents that expound upon the information in guidelines by providing more detail.

**Level 5 – Quality records –**Quality records are the documented evidence that the quality system requirements have been met.  Quality records are comprised of TMS tasks, forms, spreadsheets and logs, meeting minutes, and memos used in the day-to-day operation of the system.

<http://nrukol.blogspot.com/2010/05/qa-qc-testing-process-managers.html> - QA, QC & Testing

[http://www.softwaretestinghelp.com](http://www.softwaretestinghelp.com/) – хороший сайт про тестинг