**Тестирование. Фундаментальная теория**

Недавно был на собеседовании на Middle QA на проект, который явно превышает мои возможности. Уделил много времени тому, чего не знал вообще и мало времени повторению простой теории, а зря.

Ниже основы основ для повторения перед собеседованием для Trainee and Junior: определение тестирования, **качество**, **верификация / валидация**, цели, этапы, тест план, пункты тест плана, тест дизайн, техники тест дизайна, **traceability matrix**, test case, чек-лист, дефект, **error/deffect/failure**, баг репорт, severity vs priority, уровни тестирования, виды / типы, **подходы к интеграционному тестированию**, принципы тестирования, статическое и динамическое тестирование, исследовательское / ad-hoc тестирование, требования, жизненный цикл бага, стадии разработки ПО, decision table, qa/qc/test engineer, диаграмма связей.

Вторая часть про методологии [здесь](http://dou.ua/forums/topic/14015/).

Все замечания, корректировки и дополнения очень приветствуются.

**Тестирование программного обеспечения** — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование — это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

**Качество программного обеспечения (Software Quality)** — это совокупность характеристик программного обеспечения, относящихся к его способности удовлетворять установленные и предполагаемые потребности. [Quality management and quality assurance]

**Верификация (verification)** — это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа [IEEE]. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.  
**Валидация (validation)** — это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе [BS7925-1].  
Также можно встретить иную интерпритацию:  
Процесс оценки соответствия продукта явным требованиям (спецификациям) и есть верификация (verification), в то же время оценка соответствия продукта ожиданиям и требованиям пользователей — есть валидация (validation). Также часто можно встретить следующее определение этих понятий:  
Validation — ’is this the right specification? ’.  
Verification — ’is the system correct to specification?’.

**Цели тестирования**  
Повысить вероятность того, что приложение, предназначенное для тестирования, будет работать правильно при любых обстоятельствах.  
Повысить вероятность того, что приложение, предназначенное для тестирования, будет соответствовать всем описанным требованиям.  
Предоставление актуальной информации о состоянии продукта на данный момент.

**Этапы тестирования:**  
1. Анализ продукта  
2. Работа с требованиями  
3. Разработка стратегии тестирования  
и планирование процедур контроля качества  
4. Создание тестовой документации  
5. Тестирование прототипа  
6. Основное тестирование  
7. Стабилизация  
8. Эксплуатация

**Тест план (Test Plan)** — это документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.  
Отвечает на вопросы:  
Что надо тестировать?  
Что будете тестировать?  
Как будете тестировать?  
Когда будете тестировать?  
Критерии начала тестирования.  
Критерии окончания тестирования.

**Основные пункты тест плана**  
В стандарте IEEE 829 перечислены пункты, из которых должен (пусть — может) состоять тест-план:  
a) Test plan identifier;  
b) Introduction;  
c) Test items;  
d) Features to be tested;  
e) Features not to be tested;  
f) Approach;  
g) Item pass/fail criteria;  
h) Suspension criteria and resumption requirements;  
i) Test deliverables;  
j) Testing tasks;  
k) Environmental needs;  
l) Responsibilities;  
m) Staffing and training needs;  
n) Schedule;  
o) Risks and contingencies;  
p) Approvals.

**Тест дизайн** — это этап процесса тестирования ПО, на котором проектируются и создаются тестовые сценарии (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования.  
Роли, ответственные за тест дизайн:  
• Тест аналитик — определяет «ЧТО тестировать?»  
• Тест дизайнер — определяет «КАК тестировать?»

**Техники тест дизайна**

**• Эквивалентное Разделение (Equivalence Partitioning — EP)**. Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала — 0.

**• Анализ Граничных Значений (Boundary Value Analysis — BVA).** Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям, имеющим ограничения.

**• Причина / Следствие (Cause/Effect — CE).** Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как «Имя», «Адрес», «Номер Телефона» а затем, нажать кнопку «Добавить» — это «Причина». После нажатия кнопки «Добавить», система добавляет клиента в базу данных и показывает его номер на экране — это «Следствие».

**• Предугадывание ошибки (Error Guessing — EG).** Это когда тестировщик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы «предугадать» при каких входных условиях система может выдать ошибку. Например, спецификация говорит: «пользователь должен ввести код». Тестировщик будет думать: «Что, если я не введу код?», «Что, если я введу неправильный код?», и так далее. Это и есть предугадывание ошибки.

**• Исчерпывающее тестирование (Exhaustive Testing — ET)** — это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

**• Попарное тестирование (Pairwise Testing)** — это техника формирования наборов тестовых данных. Сформулировать суть можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

Допустим, какое-то значений (налог) для человека рассчитывается на основании его пола, возраста и наличия детей — получаем три входных параметра, для каждого из которых для тестов выбираем каким-то образом значения. Например: пол — мужской или женский; возраст — до 25, от 25 до 60, более 60; наличие детей — да или нет. Для проверки правильности расчётов можно, конечно, перебрать все комбинации значений всех параметров:

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **пол** | **возраст** | **дети** |
| **1** | мужчина | до 25 | детей нет |
| **2** | женщина | до 25 | детей нет |
| **3** | мужчина | 25-60 | детей нет |
| **4** | женщина | 25-60 | детей нет |
| **5** | мужчина | старше 60 | детей нет |
| **6** | женщина | старше 60 | детей нет |
| **7** | мужчина | до 25 | дети есть |
| **8** | женщина | до 25 | дети есть |
| **9** | мужчина | 25-60 | дети есть |
| **10** | женщина | 25-60 | дети есть |
| **11** | мужчина | старше 60 | дети есть |
| **12** | женщина | старше 60 | дети есть |

А можно решить, что нам не нужны сочетания значений всех параметров со всеми, а мы хотим только убедиться, что мы проверим все уникальные пары значений параметров. Т.е., например, с точки зрения параметров пола и возраста мы хотим убедиться, что мы точно проверим мужчину до 25, мужчину между 25 и 60, мужчину после 60, а также женщину до 25, женщину между 25 и 60, ну и женщину после 60. И точно так же для всех остальных пар параметров. И таким образом, мы можем получить гораздо меньше наборов значений (в них есть все пары значений, правда некоторые дважды):

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **пол** | **возраст** | **дети** |
| **1** | мужчина | до 25 | детей нет |
| **2** | женщина | до 25 | дети есть |
| **3** | мужчина | 25-60 | дети есть |
| **4** | женщина | 25-60 | детей нет |
| **5** | мужчина | старше 60 | детей нет |
| **6** | женщина | старше 60 | дети есть |

Такой подход примерно и составляет суть техники pairwise testing — мы не проверяем все сочетания всех значений, но проверяем все пары значений.

**Traceability matrix — Матрица соответствия требований** — это двумерная таблица, содержащая соответсвие функциональных требований (functional requirements) продукта и подготовленных тестовых сценариев (test cases). В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки.  
Матрица соответсвия требований используется QA-инженерами для валидации покрытия продукта тестами. МСТ является неотъемлемой частью тест-плана.

**Тестовый сценарий (Test Case)** — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.  
Пример:  
Action Expected Result Test Result  
(passed/failed/blocked)  
Open page «login» Login page is opened Passed

Каждый тест кейс должен иметь 3 части:  
PreConditions Список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.  
Test Case Description Список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям  
PostConditions Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста — initial state)  
Виды Тестовых Сценариев:  
Тест кейсы разделяются по ожидаемому результату на позитивные и негативные:  
• Позитивный тест кейс использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.  
• Негативный тест кейс оперирует как корректными так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

**Чек-лист (check list)** — это документ, описывающий что должно быть протестировано. При этом чек-лист может быть абсолютно разного уровня детализации. На сколько детальным будет чек-лист зависит от требований к отчетности, уровня знания продукта сотрудниками и сложности продукта.  
Как правило, чек-лист содержит только действия (шаги), без ожидаемого результата. Чек-лист менее формализован чем тестовый сценарий. Его уместно использовать тогда, когда тестовые сценарии будут избыточны. Также чек-лист ассоциируются с гибкими подходами в тестировании.

**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.

**Error** — ошибка пользователя, то есть он пытается использовать программу иным способом.  
Пример — вводит буквы в поля, где требуется вводить цифры (возраст, количество товара и т.п.).  
В качественной программе предусмотрены такие ситуации и выдаются сообщение об ошибке (error message), с красным крестиком которые.  
**Bug (defect)** — ошибка программиста (или дизайнера или ещё кого, кто принимает участие в разработке), то есть когда в программе, что-то идёт не так как планировалось и программа выходит из-под контроля. Например, когда никак не контроллируется ввод пользователя, в результате неверные данные вызывают краши или иные «радости» в работе программы. Либо внутри программа построена так, что изначально не соответствует тому, что от неё ожидается.  
**Failure** — сбой (причём не обязательно аппаратный) в работе компонента, всей программы или системы. То есть, существуют такие дефекты, которые приводят к сбоям (A defect caused the failure) и существуют такие, которые не приводят. UI-дефекты, например. Но аппаратный сбой, никак не связанный с software, тоже является failure.

**Баг Репорт (Bug Report)** — это документ, описывающий ситуацию или последовательность действий приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.  
Шапка  
Короткое описание (Summary) Короткое описание проблемы, явно указывающее на причину и тип ошибочной ситуации.  
Проект (Project) Название тестируемого проекта  
Компонент приложения (Component) Название части или функции тестируемого продукта  
Номер версии (Version) Версия на которой была найдена ошибка  
Серьезность (Severity) Наиболее распространена пятиуровневая система градации серьезности дефекта:  
• S1 Блокирующий (Blocker)  
• S2 Критический (Critical)  
• S3 Значительный (Major)  
• S4 Незначительный (Minor)  
• S5 Тривиальный (Trivial)  
Приоритет (Priority) Приоритет дефекта:  
• P1 Высокий (High)  
• P2 Средний (Medium)  
• P3 Низкий (Low)  
Статус (Status) Статус бага. Зависит от используемой процедуры и жизненного цикла бага (bug workflow and life cycle)

Автор (Author) Создатель баг репорта  
Назначен на (Assigned To) Имя сотрудника, назначенного на решение проблемы  
Окружение  
ОС / Сервис Пак и т.д. / Браузера + версия / ... Информация об окружении, на котором был найден баг: операционная система, сервис пак, для WEB тестирования — имя и версия браузера и т.д.  
...  
Описание  
Шаги воспроизведения (Steps to Reproduce) Шаги, по которым можно легко воспроизвести ситуацию, приведшую к ошибке.  
Фактический Результат (Result) Результат, полученный после прохождения шагов к воспроизведению  
Ожидаемый результат (Expected Result) Ожидаемый правильный результат  
Дополнения  
Прикрепленный файл (Attachment) Файл с логами, скриншот или любой другой документ, который может помочь прояснить причину ошибки или указать на способ решения проблемы

**Severity vs Priority**  
Серьезность (Severity) — это атрибут, характеризующий влияние дефекта на работоспособность приложения.  
Приоритет (Priority) — 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.  
Severity выставляется тестировщиком  
Priority — менеджером, тимлидом или заказчиком

Градация Серьезности дефекта (Severity)

**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 Тривиальная (Trivial)**  
Тривиальная ошибка, не касающаяся бизнес логики приложения, плохо воспроизводимая проблема, малозаметная посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Градация Приоритета дефекта (Priority)  
**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.

Уровни Тестирования

**1. Модульное тестирование (Unit Testing)**  
Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).

**2. Интеграционное тестирование (Integration Testing)**  
Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

**3. Системное тестирование (System Testing)**  
Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, 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.

**4. Операционное тестирование (Release Testing).**  
Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес модели системы. Следует учесть, что и бизнес модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.

**5. Приемочное тестирование (Acceptance Testing)**  
Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:  
• определения удовлетворяет ли система приемочным критериям;  
• вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

Виды / типы тестирования

**Функциональные виды тестирования**

• Функциональное тестирование (Functional testing)  
• Тестирование пользовательского интерфейса (GUI Testing)  
• Тестирование безопасности (Security and Access Control Testing)  
• Тестирование взаимодействия (Interoperability Testing)

**Нефункциональные виды тестирования**

• Все виды тестирования производительности:  
o нагрузочное тестирование (Performance and Load Testing)  
o стрессовое тестирование (Stress Testing)  
o тестирование стабильности или надежности (Stability / Reliability Testing)  
o объемное тестирование (Volume Testing)  
• Тестирование установки (Installation testing)  
• Тестирование удобства пользования (Usability Testing)  
• Тестирование на отказ и восстановление (Failover and Recovery Testing)  
• Конфигурационное тестирование (Configuration Testing)

**Связанные с изменениями виды тестирования**

• Дымовое тестирование (Smoke Testing)  
• Регрессионное тестирование (Regression Testing)  
• Повторное тестирование (Re-testing)  
• Тестирование сборки (Build Verification Test)  
• Санитарное тестирование или проверка согласованности/исправности (Sanity Testing)

**Функциональное тестирование** рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

**Тестирование пользовательского интерфейса (GUI Testing)** — функциональная проверка интерфейса на соответствие требованиям — размер, шрифт, цвет, consistent behavior.

**Тестирование безопасности**— это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

**Тестирование взаимодействия (Interoperability Testing)**— это функциональное тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование

**Нагрузочное тестирование** — это автоматизированное тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе.

**Стрессовое тестирование (Stress Testing)** позволяет проверить насколько приложение, и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом цели стрессового тестирования могут пересекаться с целями тестирования производительности.

**Объемное тестирование (Volume Testing).** Задачей объемного тестирования является получение оценки производительности при увеличении объемов данных в базе данных приложения

**Тестирование стабильности или надежности (Stability / Reliability Testing).** Задачей тестирования стабильности (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.

**Тестирование установки** направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.

**Тестирование удобства пользования** — это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. Сюда также входит:  
User eXperience (UX) — ощущение, испытываемое пользователем во время использования цифрового продукта, в то время как User interface — это инструмент, позволяющий осуществлять интеракцию «пользователь — веб-ресурс».

**Тестирование на отказ и восстановление (Failover and Recovery Testing)**проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

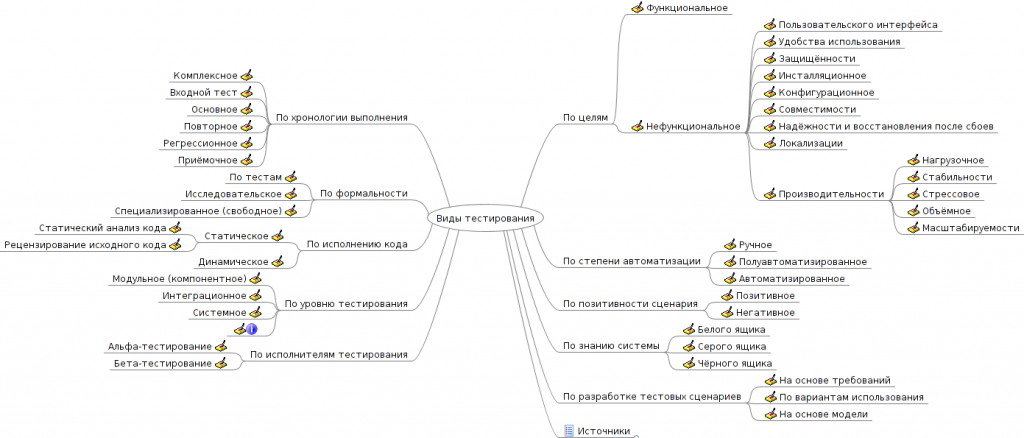
**Конфигурационное тестирование (Configuration Testing)**— специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.)

**Дымовое (Smoke)** тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции.

**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.

**Повторное тестирование** — тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.  
В чем разница между regression testing и re-testing?  
Re-testing — проверяется исправление багов  
Regression testing — проверяется то, что исправление багов, а также любые изменения в коде приложения, не повлияли на другие модули ПО и не вызвало новых багов.

**Тестирование сборки или Build Verification Test** — тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.

[](http://habrastorage.org/files/bd6/dcb/bb7/bd6dcbbb7d7c44a485b65ae29b4c0ae4.png)**Санитарное тестирование**— это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

**Подходы к интеграционному тестированию:**  
**• Снизу вверх (Bottom Up Integration)**  
Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения.  
**• Сверху вниз (Top Down Integration)**  
Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз.  
**• Большой взрыв («Big Bang» Integration)**  
Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования.

Принципы тестирования

**Принцип 1** — Тестирование демонстрирует наличие дефектов (Testing shows presence of defects)  
Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет. Тестирование снижает вероятность наличия дефектов, находящихся в программном обеспечении, но, даже если дефекты не были обнаружены, это не доказывает его корректности.

**Принцип 2**— Исчерпывающее тестирование недостижимо (Exhaustive testing is impossible)  
Полное тестирование с использованием всех комбинаций вводов и предусловий физически невыполнимо, за исключением тривиальных случаев. Вместо исчерпывающего тестирования должны использоваться анализ рисков и расстановка приоритетов, чтобы более точно сфокусировать усилия по тестированию.

**Принцип 3**— Раннее тестирование (Early testing)  
Чтобы найти дефекты как можно раньше, активности по тестированию должны быть начаты как можно раньше в жизненном цикле разработки программного обеспечения или системы, и должны быть сфокусированы на определенных целях.

**Принцип 4** — Скопление дефектов (Defects clustering)  
Усилия тестирования должны быть сосредоточены пропорционально ожидаемой, а позже реальной плотности дефектов по модулям. Как правило, большая часть дефектов, обнаруженных при тестировании или повлекших за собой основное количество сбоев системы, содержится в небольшом количестве модулей.

**Принцип 5** — Парадокс пестицида (Pesticide paradox)  
Если одни и те же тесты будут прогоняться много раз, в конечном счете этот набор тестовых сценариев больше не будет находить новых дефектов. Чтобы преодолеть этот «парадокс пестицида», тестовые сценарии должны регулярно рецензироваться и корректироваться, новые тесты должны быть разносторонними, чтобы охватить все компоненты программного обеспечения,  
или системы, и найти как можно больше дефектов.

**Принцип 6** — Тестирование зависит от контекста (Testing is concept depending)  
Тестирование выполняется по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем сайт электронной коммерции.  
**Принцип 7** — Заблуждение об отсутствии ошибок (Absence-of-errors fallacy)  
Обнаружение и исправление дефектов не помогут, если созданная система не подходит пользователю и не удовлетворяет его ожиданиям и потребностям.

**Cтатическое и динамическое тестирование**  
Статическое тестирование отличается от динамического тем, что производится без запуска программного кода продукта. Тестирование осуществляется путем анализа программного кода (code review) или скомпилированного кода. Анализ может производиться как вручную, так и с помощью специальных инструментальных средств. Целью анализа является раннее выявление ошибок и потенциальных проблем в продукте. Также к статическому тестированию относится тестирования спецификации и прочей документации.

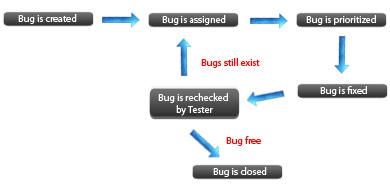
**Исследовательское / ad-hoc тестирование**  
Простейшее определение исследовательского тестирования — это разработка и выполнения тестов в одно и то же время. Что является противоположностью сценарного подхода (с его предопределенными процедурами тестирования, неважно ручными или автоматизированными). Исследовательские тесты, в отличие от сценарных тестов, не определены заранее и не выполняются в точном соответствии с планом.

Разница между ad hoc и exploratory testing в том, что теоретически, ad hoc может провести кто угодно, а для проведения exploratory необходимо мастерство и владение определенными техниками. Обратите внимание, что определенные техники это не только техники тестирования.

**Требования** — это спецификация (описание) того, что должно быть реализовано.  
Требования описывают то, что необходимо реализовать, без детализации технической стороны решения. Что, а не как.

**Требования к требованиям:**  
• Корректность  
• Недвусмысленность  
• Полнота набора требований  
• Непротиворечивость набора требований  
• Проверяемость (тестопригодность)  
• Трассируемость  
• Понимаемость

**Жизненный цикл бага**

[](http://www.zeelabs.com/images/bug_life_cycle_diagram.jpg)

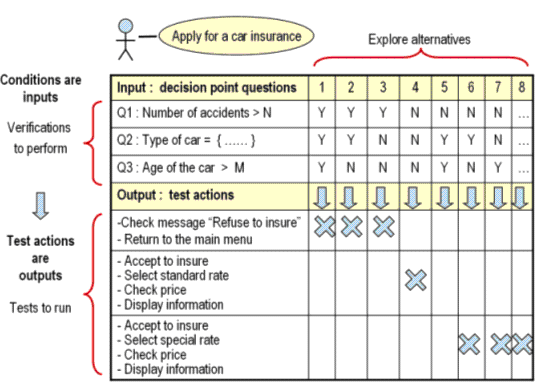
**Стадии разработки ПО** — это этапы, которые проходят команды разработчиков ПО, прежде чем программа станет доступной для широко круга пользователей. Разработка ПО начинается с первоначального этапа разработки (стадия «пре-альфа») и продолжается стадиями, на которых продукт дорабатывается и модернизируется. Финальным этапом этого процесса становится выпуск на рынок окончательной версии программного обеспечения («общедоступного релиза»).

Программный продукт проходит следующие стадии:  
• анализ требований к проекту;  
• проектирование;  
• реализация;  
• тестирование продукта;  
• внедрение и поддержка.

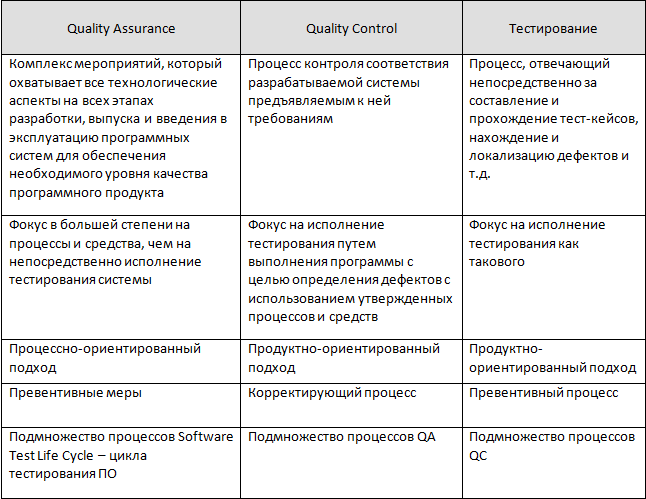
Каждой стадии разработки ПО присваивается определенный порядковый номер. Также каждый этап имеет свое собственное название, которое характеризует готовность продукта на этой стадии.

**Жизненный цикл разработки ПО:**  
• Пре-альфа  
• Альфа  
• Бета  
• Релиз-кандидат  
• Релиз  
• Пост-релиз

**Таблица принятия решений (decision table)** — великолепный инструмент для упорядочения сложных бизнес требований, которые должны быть реализованы в продукте. В таблицах решений представлен набор условий, одновременное выполнение которых должно привести к определенному действию.

[](http://www.interface.ru/iarticle/img/1170_3.bmp%09)

**QA/QC/Test Engineer**

[](http://qalight.com.ua/images/Screenshot_2.png)

Таким образом, мы можем построить модель иерархии процессов обеспечения качества: Тестирование — часть QC. QC — часть QA.

**Диаграмма связей** — это инструмент управления качеством, основанный на определении логических взаимосвязей между различными данными. Применяется этот инструмент для сопоставления причин и следствий по исследуемой проблеме.  
[](http://sixsigmaonline.ru/imgs/001/a0194.png)

Источники: [www.protesting.ru](http://www.protesting.ru/), [bugscatcher.net](http://bugscatcher.net/), [qalight.com.ua](http://qalight.com.ua/), [thinkingintests.wordpress.com](https://thinkingintests.wordpress.com/), книга ISTQB, [www.quizful.net](http://www.quizful.net/), [bugsclock.blogspot.com](http://bugsclock.blogspot.com/), [www.zeelabs.com](http://www.zeelabs.com/), [devopswiki.net](http://devopswiki.net/), [hvorostovoz.blogspot.com](http://hvorostovoz.blogspot.com/).

Ресурсы рекомендованные в комментах Sofiya Novachenko: [istqbexamcertification.com](http://istqbexamcertification.com/) [www.testingexcellence.com](http://www.testingexcellence.com/)

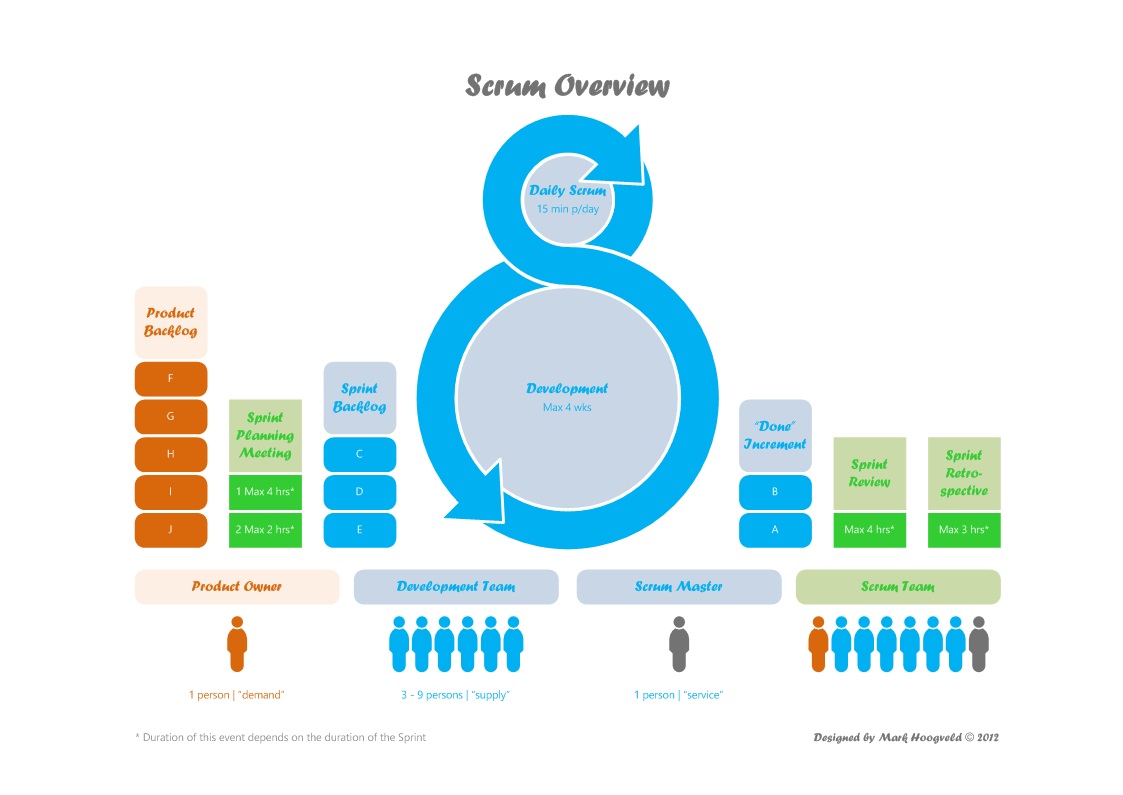
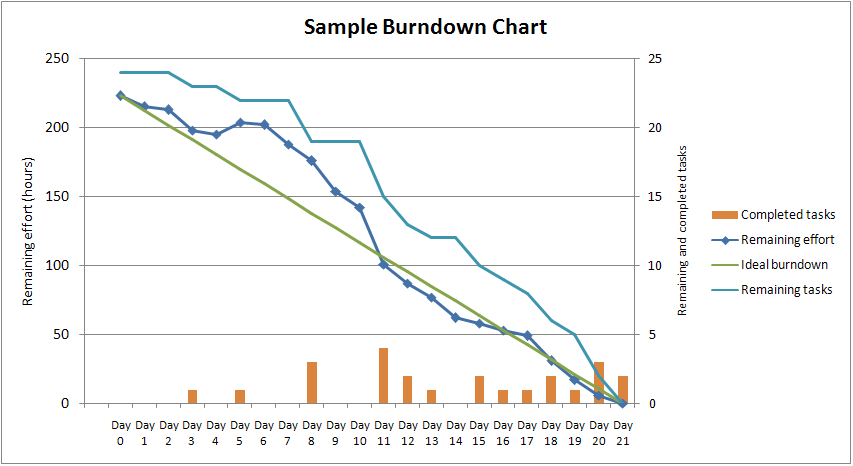
**Тестирование. Фундаментальная теория. Часть 2 — Методологии разработки ПО**

Наконец-то дошли руки до второй шпоры. Первая лежит [здесь](http://dou.ua/forums/topic/13389/). Эта будет посвящена методологиям разработки и так же будет полезна новичкам в тестировании, ибо это так же часто спрашивают на собеседованиях.  
И, как всегда, критика, уточнения, дополнения are extremely welcome.

**Виды методологий разработки ПО:**

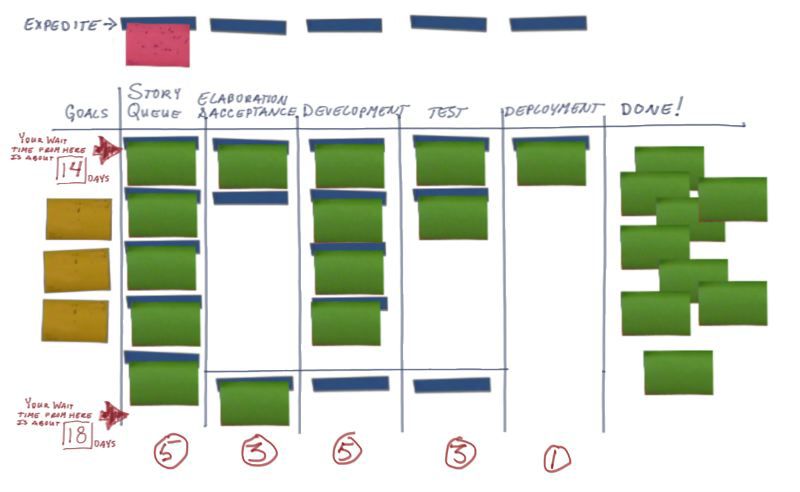
• **Каскадная или поэтапная разработка** (в некоторых источниках её называют «водопадной моделью») — процесс создания программного обеспечения представляет собой поток, последовательно проходящий фазы анализа, проектирования, реализации, тестирования, интеграции и поддержки. Так обычно строится работа над крупными проектами с длительным сроком внедрения.  
• **Итеративная или инкрементная (эволюционная) модель** приращения продукта позволяет параллельно выполнять ряд задач с непрерывным анализом результатов и корректировкой предыдущих этапов работы. Это более «скоростная» разработка для большого штата квалифицированных программистов.  
• **Спиральная методика** характеризуется прохождением проектом повторяющегося цикла в каждой фазе развития: планирование — реализация — проверка — оценка (англ. plan-do-check-act cycle). Так обычно создаются проекты, с окончательно не сформированным видением результата, либо требующие ультрасрочного внедрения по этапам.  
• **Гибкая методология Agile-разработки** — т.н. быстрая разработка без ущерба качеству, когда во главу угла ставится работающий продукт, а не его документация. Наиболее современный неформализованный подход к созданию ПО, в процессе которого реагирование на изменения ценятся выше строгого следования плану. Для молодых стремительно развивающихся проектов, которые с каждой итерацией программного обеспечения, по сути, готовы к его релизу.

**SCRUM**

[](http://calvinx.com/wp-content/uploads/2014/05/scrum-overview-mark-hoogveld.jpg)  
**Scrum (Скрам)** — это не аббревиатура, этот термин взят из регби, который обозначает схватку вокруг мяча.  
Сам термин Scrum, я бы определила так — это методология управления проектами, которая построена на принципах тайм-менеджмета. Основной ее особенностью является вовлеченность в процесс всех участников, причем у каждого участника есть своя определенная роль. Суть в том, что не только команда работает над решением задачи, но все те, кому интересно решение задачи, не просто поставили ее и расслабились, а постоянно «работают» с командой, и эта работа не означает только постоянный контроль.  
Основные термины, которые используются в методологии:  
**Владелец продукта (Product owner)** — человек, который имеет непосредственный интерес в качественном конечном продукте, он понимает, как это продукт должен выглядеть/работать. Этот человек не работает в команде, он работает на стороне заказчика/клиента (это может быть как другая компания, так и другой отдел), но этот человек работает с командой. И это тот человек, который расставляет приоритеты для задач.  
**Scrum-мастер** — это человек, которого можно назвать руководителем проекта, хотя это не совсем так. Главное, что это человек, «зараженный Scrum-бациллой» на столько, что несет ее как своей команде, так и заказчику, и соответственно следит за тем, чтобы все принципы Scrum соблюдались.  
**Scrum-команда** — это команда, которая принимает все принципы Scrum и готова с ними работать.  
**Спринт** — отрезок времени, который берется для выполнения определенного (ограниченного) списка задач. Рекомендуется брать 2-4 недели (длительность определяется командой один раз).  
**Бэклог (backlog)** — это список всех работ. Можно сказать, что это ежедневник общего пользования   
Различают 2 вида бэклогов: Product-бэклог и спринт-бэклог.  
**Product-бэклог** — это полный список всех работ, при реализации которых мы получим конечный продукт.  
**Спринт-бэклог** — это список работ, который определила команда и согласовала с Владельцем продукта, на ближайший отчетный период (спринт). Задания в спринт-бэклог берутся из product-бэклога.  
**Планирование спринта** — это совещание, на котором присутствуют все (команда, Scrum-мастер, Владелец продукта). В течение этого совещания Владелец продукта определяет приоритеты заданий, которые он хотел бы увидеть выполнеными по истечении спринта. Команда оценивает по времени, сколько из желаемого они могут выполнить. В итоге получается список заданий, который не может меняться в течение спринта и к концу спринта должен быть полностью выполнен.  
Пример работы PR-агентства. Как бы это могло выглядеть, если бы они работали по Scrum.  
Компания клиент «Икс» хочет провести через 2 месяца масштабное мероприятие для своих партнеров и журналистов. Услуги по организации такого мероприятия компания «Икс» заказала у агентства «Зет». Компанию «Икс» представляет PR-менеджер, который отвечает за организацию мероприятия со стороны клиента. В терминологии Scrum — этот человек называется Владелец продукта. Со стороны агентства за организацию мероприятия отвечает account-менеджер (Scrum-мастер), в подчинении которого находится команда (Scrum-команда). На совместном совещании (планировании спринта) компания и агентство решают, что они будут отчитываться-планировать каждые 2 недели (длина спринта). На первые 2 недели они запланировали список задач (спринт-бэклог), однако команда оценила, что не все из этого списка они успеют выполнить. Тогда PR-менеджер (он же Владелец продукта), говорит какие из этого списка задач более приоритетные на ближайшие 2 недели, после чего команда берется за выполнение заданий. Единственное что здесь должно быть учтено, что на момент планирования первого спринта должен быть спланирован весь список заданий на 2 месяца (product-бэклог), чтобы не получилось так, что к моменту проведения мероприятия что-то не выполнено.  
**Жизненный цикл спринта**  
**Планирование спринта**  
В начале каждого спринта проводится планирование спринта. В планировании спринта участвуют заказчики, пользователи, менеджмент, Product Owner, Скрам Мастер и команда.  
Планирование спринта состоит из двух последовательных митингов.  
Планирование спринта, митинг первый  
Участники: команда, Product Owner, Scrum Master, пользователи, менеджемент  
Цель: Определить цель спринта (Sprint Goal) и Sprint Backlog -функциональность, которая будет разработана в течение следующего спринта для достижения цели спринта.  
Артефакт: Sprint Backlog  
Планирование спринта, митинг второй  
Участники: Скрам Мастер, команда  
Цель: определить, как именно будет разрабатываться определенная функциональность для того, чтобы достичь цели спринта. Для каждого элемента Sprint Backlog определяется список задач и оценивается их продолжительность.  
Артефакт: в Sprint Backlog появляются задачи  
Если в ходе спринта выясняется, что команда не может успеть сделать запланированное на спринт, то Скрам Мастер, Product Owner и команда встречаются и выясняют, как можно сократить scope работ и при этом достичь цели спринта.  
**Остановка спринта (Sprint Abnormal Termination)**  
Остановка спринта производится в исключительных ситуациях. Спринт может быть остановлен до того, как закончатся отведенные 30 дней. Спринт может остановить команда, если понимает, что не может достичь цели спринта в отведенное время. Спринт может остановить Product Owner, если необходимость в достижении цели спринта исчезла.  
После остановки спринта проводится митинг с командой, где обсуждаются причины остановки спринта. После этого начинается новый спринт: производится его планирование и стартуются работы.  
**Daily Scrum Meeting**  
Этот митинг проходит каждое утро в начале дня. Он предназначен для того, чтобы все члены команды знали, кто и чем занимается в проекте. Длительность этого митинга строго ограничена и не должна превышать 15 минут. Цель митинга — поделиться информацией. Он не предназначен для решения проблем в проекте. Все требующие специального обсуждения вопросы должны быть вынесены за пределы митинга.  
Скрам митинг проводит Скрам Мастер. Он по кругу задает вопросы каждому члену команды  
• Что сделано вчера?  
• Что будет сделано сегодня?  
• С какими проблемами столкнулся?  
Скрам Мастер собирает все открытые для обсуждения вопросы в виде Action Items, например в формате что/кто/когда, например  
• Обсудить проблему с отрисовкой контрола  
• Петя и Вася  
• Сразу после скрама  
**Диаграмма сгорания задач (Burndown chart)**  
[](https://upload.wikimedia.org/wikipedia/commons/0/05/SampleBurndownChart.png)

Диаграмма, показывающая количество сделанной и оставшейся работы. Обновляется ежедневно с тем, чтобы в простой форме показать подвижки в работе над спринтом. График должен быть общедоступен.  
Существуют разные виды диаграммы:  
• диаграмма сгорания работ для спринта — показывает, сколько уже задач сделано и сколько ещё остаётся сделать в текущем спринте.  
• диаграмма сгорания работ для выпуска проекта — показывает, сколько уже задач сделано и сколько ещё остаётся сделать до выпуска продукта (обычно строится на базе нескольких спринтов).  
Ретроспектива   
В конце каждого Спринта, Скрам Команда собирается на **Ретроспективу**. Цель Ретроспективы пересмотреть качество существующих процессов, взаимоотношения людей и применяемые инструменты. Команда определяет, что прошло хорошо, а что не очень, а также выявляет потенциальные возможности для улучшений. Они создают план улучшений на будущее.

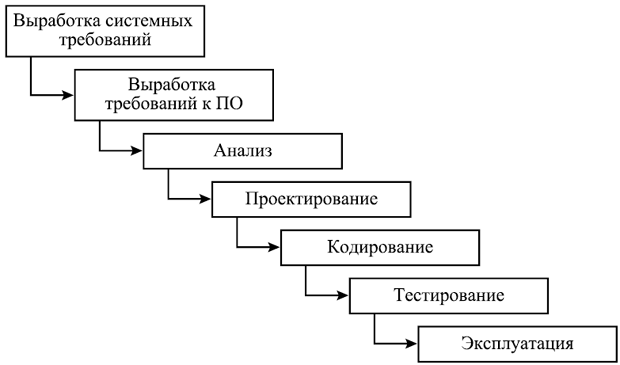
**Канбан**

Термин Канбан имеет дословный перевод: «Кан» значит видимый, визуальный, и «бан» значит карточка или доска.  
На заводах Тойота карточки Канбан используются повсеместно для того, чтобы не загромождать склады и рабочие места заранее созданными запчастями. Например, представьте, что вы ставите двери на Тойоты Короллы. У вас около рабочего места находится пачка из 10 дверей. Вы их ставите одну за другой на новые машины и, когда в пачке остается 5 дверей, то вы знаете, что пора заказать новые двери. Вы берете карточку Канбан, пишете на ней заказ на 10 дверей и относите ее тому, кто делает двери. Вы знаете, что он их сделает как раз к тому моменту, как у вас закончатся оставшиеся 5 дверей. И именно так и происходит — когда вы ставите последнюю дверь, прибывает пачка из 10 новых дверей. И так постоянно — вы заказываете новые двери только тогда, когда они вам нужны.  
А теперь представьте, что такая система действует на всём заводе. Нигде нет складов, где запчасти лежат неделями и месяцами. Все работают только по запросу и производят именно столько запчастей, сколько запрошено. Если вдруг заказов стало больше или меньше — система сама легко подстраивается под изменения.  
Основная задача карт Канбан в этой системе — это уменьшать количество «выполняющейся в данный момент работы» (work in progress).  
Например, на всю производственную линию может быть выделено ровно 10 карточек для дверей. Это значит, что в каждый момент времени на линии не будет больше 10 готовых дверей. Когда заказывать новые двери и сколько — это задача для того, кто их устанавливает. Только он знает свои потребности, и только он может помещать заказы производителю дверей, но он всегда ограничен числом 10.  
Этот метод Бережливого производства (Lean manufacturing) был придуман в Тойоте и сейчас многие производственные компании по всему миру его внедряют или уже внедрили.  
Но это всё относится к производству, а не к разработке программного обеспечения.  
А что же такое Канбан разработка применительно к ПО, и чем она отличается от других гибких методологий, буть то SCRUM или XP?  
Во-первых, нужно сразу понять, что Канбан — это не конкретный процесс, а система ценностей. Как, впрочем, и SCRUM с XP. Это значит, что никто вам не скажет что и как делать по шагам.  
Во-вторых, весь Канбан можно описать одной простой фразой — «Уменьшение выполняющейся в данный момент работы (work in progress)».  
В-третьих, Канбан — это даже еще более «гибкая» методология, чем SCRUM и XP. Это значит, что она не подойдет всем командам и для всех проектов. И это также значит, что команда должна быть еще более готовой к гибкой работе, чем даже команды, использующие SCRUM и XP.  
**Разница между Канбан и SCRUM:**  
— В Канбан нет таймбоксов ни на что (ни на задачи, ни на спринты)  
— В Канбан задачи больше и их меньше  
— В Канбан оценки сроков на задачу опциональные или вообще их нет  
— В Канбан «скорость работы команды» отсутствует и считается только среднее время на полную реализацию задачи  
Канбан разработка отличается от SCRUM в первую очередь ориентацией на задачи. Если в SCRUM основная ориентация команды — это успешное выполнение спринтов (надо признать, что это так), то в Канбан на первом месте задачи.  
Спринтов никаких нет, команда работает над задачей с самого начала и до завершения. Деплоймент задачи делается тогда, когда она готова. Презентация выполненной работы — тоже. Команда не должна оценивать время на выполнение задачи, ибо это имеет мало смысла и почти всегда ошибочно вначале.  
Если менеджер верит команде, то зачем иметь оценку времени? Задача менеджера — это создать приоритизированный пул задач, а задача команды — выполнить как можно больше задач из этого пула. Всё. Никакого контроля не нужно. Всё, что нужно от менеджера — это добавлять задачи в этот пул или менять им приоритет. Именно так он управляет проектом.  
Команда для работы использует Канбан-доску. Например, она может выглядеть так:  
[](http://habrastorage.org/files/2b5/ec1/918/2b5ec19187ef4c47821de19248fa63b1.jpg)

Столбцы слева направо:  
**Цели проекта:**  
Необязательный, но полезный столбец. Сюда можно поместить высокоуровневые цели проекта, чтобы команда их видела и все про них знали. Например, «Увеличить скорость работы на 20%» или «Добавить поддержку Windows 7».  
**Очередь задач:**  
Тут хранятся задачи, которые готовы к тому, чтобы начать их выполнять. Всегда для выполнения берется верхняя, самая приоритетная задача и ее карточка перемещается в следующий столбец.  
**Проработка дизайна:**  
этот и остальные столбцы до «Закончено» могут меняться, т.к. именно команда решает, какие шаги проходит задача до состояния «Закончено».  
Например, в этом столбце могут находиться задачи, для которых дизайн кода или интерфейса еще не ясен и обсуждается. Когда обсуждения закончены, задача передвигается в следующий столбец.  
**Разработка:**  
Тут задача висит до тех пор, пока разработка фичи не завершена. После завершения она передвигается в следующий столбец.  
Или, если архитектура не верна или не точна — задачу можно вернуть в предыдущий столбец.  
**Тестирование:**  
В этом столбце задача находится, пока она тестируется. Если найдены ошибки — возвращается в Разработку. Если нет — передвигается дальше.  
**Деплоймент:**  
У всех проектов свой деплоймент. У кого-то это значит выложить новую версию продукта на сервер, а у кого-то — просто закомитить код в репозиторий.  
**Закончено:**  
Сюда стикер попадает только тогда, когда все работы по задаче закончены полностью.

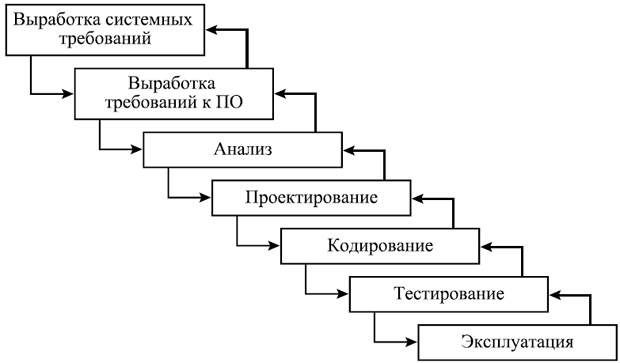
В любой работе случаются срочные задачи. Запланированные или нет, но такие, которые надо сделать прямо сейчас. Для таких можно выделить специальное место (на картинке отмечено, как «Expedite»). В Expedite можно поместить одну срочную задачу и команда должна начать ее выполнять немедленно и завершить как можно быстрее. Но может быть только одна такая задача! Если появляется еще одна — она должна быть добавлена в «Очередь задач».  
А теперь самое важное. Видите цифры под каждым столбцом? Это число задач, которые могут быть одновременно в этих столбцах. Цифры подбираются экспериментально, но считается, что они должны зависеть от числа разработчиков в команде.  
Например, если вы имеете 8 программистов в команде, то в строку «Разработка» вы можете поместить цифру 4. Это значит, что одновременно программисты будут делать не более 4-х задач, а значит у них будет много причин для общения и обмена опытом. Если вы поставите туда цифру 2, то 8 программистов, занимающихся двумя задачами, могут заскучать или терять слишком много времени на обсуждениях. Если поставить 8, то каждый будет заниматься своей задачей и некоторые задачи будут задерживаться на доске надолго, а ведь главная задача Канбан — это уменьшение времени прохождения задачи от начала до стадии готовности.  
Никто не даст точный ответ, какие должны быть эти лимиты, но попробуйте для начала разделить число разработчиков на 2 и посмотреть, как это работает в вашей команде. Потом эти числа можно подогнать под вашу команду.  
Под «разработчиками» я понимаю не только программистов, но и других специалистов. Например, для столбца «Тестирование» разработчики — это тестеры, т.к. тестирование — это их обязаность.

**Каскадная модель (waterfall)**

[](http://fresh2l.com/sites/default/files/dev-cycle-waterfall.gif)

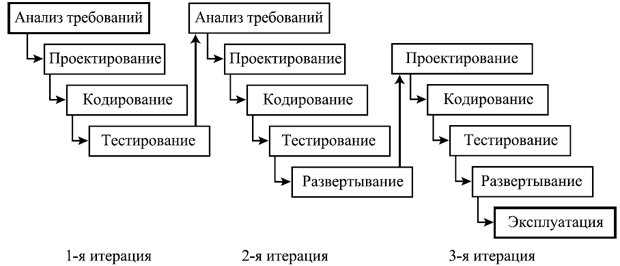
— высокий уровень формализации процессов;  
— большое количество документации;  
— жесткая последовательность этапов жизненного цикла без возможности возврата на предыдущий этап.  
Минусы  
• Watrefall проект должен постоянно иметь актуальную документацию. Обязательная актуализация проектной документации. Избыточная документация  
• Очень не гибкая методологии  
• Может создать ошибочное впечатление о работе над проектом (например фраза «45% выполнено» не несёт за собой никакой полезной информации, а является всего лишь инструментов для менеджера проекта)  
• У Заказчика нет возможности ознакомиться с системой заранее и даже с «Пилотом» системы  
• У Пользователя нет возможности привыкать к продукту постепенно  
• Все требования должны быть известны в начале жизненного цикла проекта  
• Возникает необходимость в жёстком управлении и регулярном контроле, иначе проект быстро выйдет из графиков  
• Отсутствует возможность учесть переделку, весь проект делается за один раз  
**Плюсы**  
• Высокая прозрачность разработки и фаз проекта  
• Чёткая последовательность  
• Стабильность требований  
• Строгий контроль менеджмента проекта  
• Облегчает работу по составлению плана проекта и сбора команды проекта  
• Хорошо определяет процедуру по контроля качества

**«Водоворот» или каскадная модель с промежуточным контролем**

В этой модели предусмотрен промежуточный контроль за счет обратных связей. Но это достоинство порождает и недостатки. Затраты на реализацию проекта при таком подходе возрастают практически в 10 раз. Эта модель, как вы уже поняли, является незначительной модификацией предыдущей и относится к первой группе.  
[](http://fresh2l.com/sites/default/files/dev-cycle-whirlpool.gif)

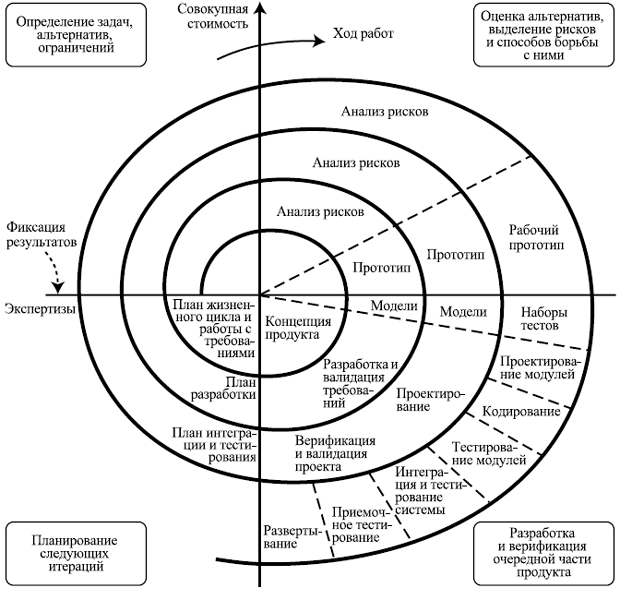
При реальной работе в соответствии с моделью, допускающей движение только в одну сторону, обычно возникают проблемы при обнаружении недоработок и ошибок, сделанных на ранних этапах. Но еще более тяжело иметь дело с изменениями окружения, в котором разрабатывается ПО (это могут быть изменения требований, смена подрядчиков, изменения политик разрабатывающей или эксплуатирующей организации, изменения отраслевых стандартов, появление конкурирующих продуктов и пр.).

**Итеративная модель**

Итеративные или инкрементальные модели (известно несколько таких моделей) предполагают разбиение создаваемой системы на набор кусков, которые разрабатываются с помощью нескольких последовательных проходов всех работ или их части.  
[](http://fresh2l.com/sites/default/files/dev-cycle-iter.gif)

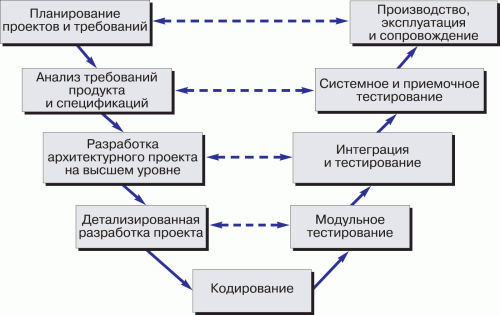
Каскадная модель с возможностью возвращения на предшествующий шаг при необходимости пересмотреть его результаты, становится итеративной.  
Итеративный процесс предполагает, что разные виды деятельности не привязаны намертво к определенным этапам разработки, а выполняются по мере необходимости, иногда повторяются, до тех пор, пока не будет получен нужный результат.  
Вместе с гибкостью и возможностью быстро реагировать на изменения, итеративные модели привносят дополнительные сложности в управление проектом и отслеживание его хода. При использовании итеративного подхода значительно сложнее становится адекватно оценить текущее состояние проекта и спланировать долгосрочное развитие событий, а также предсказать сроки и ресурсы, необходимые для обеспечения определенного качества результата.

**Спиральная модель жизненного цикла программного обеспечения**

Данная модель прекрасно сочетает в себе постадийное прототипирование и проектирование. И из восходящей и нисходящей концепций в эту модель было взято все лучшее.  
[](http://fresh2l.com/sites/default/files/dev-cycle-spiral.gif)

**Преимущества модели:**  
1. Результат достигается в кратчайшие сроки.  
2. Конкурентоспособность достаточно высокая.  
3. При изменении требований, не придется начинать все с «нуля».  
Но у этой модели есть один существенный недостаток: невозможность регламентирования стадий выполнения.

**V модель — разработка через тестирование**

Данная модель имеет более приближенный к современным методам алгоритм, однако все еще имеет ряд недостатков. Является одной из основных практик экстремального программирования и предполагает регулярное тестирование продукта во время разработки.  
[](http://fresh2l.com/sites/default/files/dev-cycle-v.png)

V-модель обеспечивает поддержку в планировании и реализации проекта. В ходе проекта ставятся следующие задачи:  
• Минимизация рисков: V-образная модель делает проект более прозрачным и повышает качество контроля проекта путём стандартизации промежуточных целей и описания соответствующих им результатов и ответственных лиц. Это позволяет выявлять отклонения в проекте и риски на ранних стадиях и улучшает качество управления проектов, уменьшая риски.  
• Повышение и гарантии качества: V-Model — стандартизованная модель разработки, что позволяет добиться от проекта результатов желаемого качества. Промежуточные результаты могут быть проверены на ранних стадиях. Универсальное документирование облегчает читаемость, понятность и проверяемость.  
• Уменьшение общей стоимости проекта: Ресурсы на разработку, производство, управление и поддержку могут быть заранее просчитаны и проконтролированы. Получаемые результаты также универсальны и легко прогнозируются. Это уменьшает затраты на последующие стадии и проекты.  
• Повышение качества коммуникации между участниками проекта: Универсальное описание всех элементов и условий облегчает взаимопонимание всех участников проекта. Таким образом, уменьшаются неточности в понимании между пользователем, покупателем, поставщиком и разработчиком.

**Модель на основе разработки прототипа**

Данная модель основывается на разработке прототипов и прототипирования продукта и относится ко второй группе.  
Прототипирование используется на ранних стадиях жизненного цикла программного обеспечения:  
o Прояснить не ясные требования (прототип UI)  
o Выбрать одно из ряда концептуальных решений (реализация сцинариев)  
o Проанализировать осуществимость проекта  
Классификация протопипов:  
o Горизонтальные прототипы — моделирует исключительно UI не затрагивая логику обработки и базу данных.  
o Вертикальные прототипы — проверка архитектурных решений.  
o Одноразовые прототипы — для быстрой разработки.  
o Эволюционные прототипы — первое приближение эволюционной системы.

**Модель Хаоса**

Вкратце Стратегия хаоса — это стратегия разработки программного обеспечения основанная на модели хаоса. Главное правило — это,всегда решать наиболее важную задачу первой.

**Экстремальное программирование (XP)**

Двенадцать основных приёмов экстремального программирования (по первому изданию книги Extreme programming explained) могут быть объединены в четыре группы:  
• Короткий цикл обратной связи (Fine-scale feedback)  
• Разработка через тестирование (Test-driven development)  
• Игра в планирование (Planning game)  
• Заказчик всегда рядом (Whole team, Onsite customer)  
• Парное программирование (Pair programming)  
• Непрерывный, а не пакетный процесс  
• Непрерывная интеграция (Continuous integration)  
• Рефакторинг (Design improvement, Refactoring)  
• Частые небольшие релизы (Small releases)  
• Понимание, разделяемое всеми  
• Простота (Simple design)  
• Метафора системы (System metaphor)  
• Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)  
• Стандарт кодирования (Coding standard or Coding conventions)  
• Социальная защищенность программиста (Programmer welfare):  
• 40-часовая рабочая неделя (Sustainable pace, Forty-hour week)

**RATIONAL UNIFIED PROCESS (RUP)**

 — методология разработки программного обеспечения, созданная компанией Rational Software.   
[](https://upload.wikimedia.org/wikipedia/commons/1/19/Development-iterative.png)

В основе методологии лежат 6 основных принципов:

• компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта;  
• работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам;  
• ранняя идентификация и непрерывное устранение возможных рисков;  
• концентрация на выполнении требований заказчиков к исполняемой программе;  
• ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки;  
• постоянное обеспечение качества на всех этапах разработки проекта.

Использование методологии RUP направлено на итеративную модель разработки. Особенность методологии состоит в том, что степень формализации может меняться в зависимости от потребностей проекта. Можно по окончании каждого этапа и каждой итерации создавать все требуемые документы и достигнуть максимального уровня формализации, а можно создавать только необходимые для работы документы, вплоть до полного их отсутствия. За счет такого подхода к формализации процессов методология является достаточно гибкой и широко популярной. Данная методология применима как в небольших и быстрых проектах, где за счет отсутствия формализации требуется сократить время выполнения проекта и расходы, так и в больших и сложных проектах, где требуется высокий уровень формализма, например, с целью дальнейшей сертификации продукта. Это преимущество дает возможность использовать одну и ту же команду разработчиков для реализации различных по объему и требованиям.

**MICROSOFT SOLUTIONS FRAMEWORK**

 — методология разработки программного обеспечения, предложенная корпорацией Microsoft. MSF опирается на практический опыт Microsoft и описывает управление людьми и рабочими процессами в процессе разработки решения.

Базовые концепции и принципы модели процессов MSF:

• единое видение проекта — все заинтересованные лица и просто участники проекта должны чётко представлять конечный результат, всем должна быть понятна цель проекта;  
• управление компромиссами — поиск компромиссов между ресурсами проекта, календарным графиком и реализуемыми возможностями;  
• гибкость — готовность к изменяющимся проектным условиям;  
• концентрация на бизнес-приоритетах — сосредоточенность на той отдаче и выгоде, которую ожидает получить потребитель решения;  
• поощрение свободного общения внутри проекта;  
• создание базовых версии — фиксация состояния любого проектного артефакта, в том числе программного кода, плана проекта, руководства пользователя, настройки серверов и последующее эффективное управление изменениями, аналитика проекта.  
MSF предлагает проверенные методики для планирования, проектирования, разработки и внедрения успешных IT-решений. Благодаря своей гибкости, масштабируемости и отсутствию жестких инструкций MSF способен удовлетворить нужды организации или проектной группы любого размера. Методология MSF состоит из принципов, моделей и дисциплин по управлению персоналом, процессами, технологическими элементами и связанными со всеми этими факторами вопросами, характерными для большинства проектов.  
Источники: [www.web-pharus.ru](http://www.web-pharus.ru/), [habrahabr.ru](http://habrahabr.ru/), [youmanager.pro](http://youmanager.pro/), [fresh2l.com](http://fresh2l.com/), [ru.wikipedia.org](https://ru.wikipedia.org/), [2programmer.ru/tehnolog1?start=5](http://2programmer.ru/tehnolog1?start=5), [tim.com.ua](http://tim.com.ua/).  
Nice to read: [www.scrumalliance.org/...​ns/Core-Scrum-Russian.pdf](https://www.scrumalliance.org/scrum/media/ScrumAllianceMedia/Files%20and%20PDFs/Why%20Scrum/Core%20Scrum%20Translations/Core-Scrum-Russian.pdf)