[Scrum teams](https://www.aha.io/roadmapping/guide/agile/what-are-scrum-roles) are self-organizing — that decide who works on what and when the work is done. But how do you accomplish this? To stay aligned and on track to meet goals, scrum teams hold four events related to sprints:

1. Sprint planning meeting //
2. [Daily standup](https://www.aha.io/roadmapping/guide/agile/what-is-a-daily-standup)
3. [Sprint reviews](https://www.aha.io/roadmapping/guide/agile/sprint-review)
4. [Sprint retrospective](https://www.aha.io/roadmapping/guide/agile/what-is-an-agile-retrospective)
5. **Sprint planning meeting:** В нём принимают участие Project Owner, Scrum Master и вся developing team. Иногда происходит и участие внешних заинтересованных сторон, однако это бывает редко. Во время Sprint Planning Meeting [Product Owner](https://scrum-time.com/infobase/product-owner.php) описывает наиболее приоритетные задачи команды. Команда в это время задаёт достаточное количество вопросов, чтобы более точно оценить и распределить задачи, которые будут решаться во время [Sprint](https://scrum-time.com/infobase/scrum-sprint.php)[[1]](#footnote-1). Project Owner не должен описывать каждый элемент [Product Backlog](https://scrum-time.com/infobase/product-backlog.php). Для Product Owner хорошим ориентиром будет приход на Sprint Planning Meeting и разговор о задачах, которые в сумме будут распределены на два спринта. Если, скажем, команда будет брать 5 задач на текущий Sprint, то разговор будет о топ-10 задачах из всего Backlog.
6. **Daily standup meeting**: это короткие собрания, которые проводятся **ежедневно** в гибких командах. На этих собраниях каждый участник команды тезисно рассказывает о своих рабочих задачах. Вот три ключевых вопроса, на которые отвечают на стендапах:

**Что я сделал вчера?**

**Что я буду делать сегодня?**

**Какие сложности могут возникнуть/возникают?**

Стендапы помогают поддерживать прозрачность процессов, быстро обмениваться важными новостями о работе и оперативно решать спорные моменты в работе коллег. [Это также помогает сотрудникам почувствовать свою причастность к команде, сфокусироваться на своих задачах и лучше понять, кто за какие задачи отвечает в команде1](https://weeek.net/ru/blog/kak-ezednevnye-stendapy-pomogaut-v-upravlenii-komandoj).

1. **Sprint Review** – это важный этап в методологии **Scrum**, который происходит после завершения каждого спринта. Давайте разберемся, что это такое и как он работает.
2. [**Что такое обзор спринта?** Проще говоря, обзор спринта – это краткая оценка результатов спринта и любые корректировки, которые необходимо внести в дальнейшую работу для обеспечения успеха команды1](https://quasa.io/ru/media/sprint-reviews-kak-oni-rabotayut-i-kak-ih-provodit). Это собрание позволяет заинтересованным сторонам сотрудничать и оценивать свои достижения в течение спринта.
3. **Как проходит обзор спринта?** Во время обзора команда демонстрирует выполненную работу заинтересованным сторонам. Они рассматривают задачи, которые были выполнены, и те, которые остались незавершенными. [Также обсуждаются все потенциальные проблемы, которые могли возникнуть в ходе спринта, и рассматриваются решения этих проблем1](https://quasa.io/ru/media/sprint-reviews-kak-oni-rabotayut-i-kak-ih-provodit).
4. **Цель обзора спринта:**
   * Создать ответственность среди участников разработки.
   * [Информировать о будущих спринтах, предоставляя полезные идеи и ресурсы для дальнейшей работы](https://quasa.io/ru/media/sprint-reviews-kak-oni-rabotayut-i-kak-ih-provodit)[2](https://bing.com/search?q=%D1%87%D1%82%D0%BE+%D1%82%D0%B0%D0%BA%D0%BE%D0%B5+Sprint+reviews).

Обзор спринта – это важнейшая рабочая сессия, которая помогает команде оценить свой прогресс и спланировать следующие шаги в достижении целей. 🚀

1. **Sprint Retrospective** (Ретроспектива спринта) — это событие в методологии **Scrum**, которое происходит в конце каждого спринта. Его цель заключается в планировании способов **повышения качества и эффективности**. Во время ретроспективы команда Scrum анализирует, как прошел предыдущий спринт, обращая внимание на взаимодействие между участниками, процессы, инструменты и их определение готовности (Definition of Done). Затем команда Scrum выявляет наиболее полезные изменения для повышения своей эффективности. Самые значимые улучшения рассматриваются как можно скорее и могут даже быть добавлены в **backlog** спринта на следующий период. Ретроспектива спринта завершает сам спринт и ограничивается максимум 3 часами для спринта продолжительностью в один месяц. Для более коротких спринтов это событие обычно занимает меньше времени. Во время ретроспективы команда обсуждает следующие вопросы:
2. Что прошло хорошо в спринте?
3. Что можно улучшить?
4. Над чем мы обязуемся работать, чтобы улучшить следующий спринт?

Во время каждой ретроспективы спринта команда Scrum планирует способы повышения качества продукта, улучшая рабочие процессы или адаптируя определение “готовности”, если это соответствует стандартам продукта или организации.

**Daily Stand Up meeting insists of:**

* What did I work on yesterday?
* What am I working on today?
* What issues are blocking me?

**Product Backlog**

**Бэклог продукта** — это перечень рабочих задач, расположенных в порядке важности, для команды разработчиков. Его составляют на основе дорожной карты и требований в ней. [Наиболее важные задачи расположены в начале бэклога продукта, чтобы команда понимала, какую работу следует выполнить в первую очередь1](https://www.atlassian.com/ru/agile/scrum/backlogs)[2](https://bing.com/search?q=%D1%87%D1%82%D0%BE+%D1%82%D0%B0%D0%BA%D0%BE%D0%B5+Product+Backlog).

**Бэклог продукта** — это упорядоченный и постоянно обновляемый список всего, что планируется сделать для создания и улучшения продукта. [Этот артефакт Скрама является единственным источником работы для Скрам-команды3](https://scrumtrek.ru/blog/agile-scrum/scrum-glossary/3758/product-backlog/). **Project owner** составляет бэклог на основе дорожной карты и требований. [Команда разработчиков выбирает задачи из бэклога продукта, когда у нее есть необходимые ресурсы, выполняя их непрерывно (Kanban) или итерациями (Scrum)](https://www.atlassian.com/ru/agile/scrum/backlogs).

Бэклог продукта состоит из двух столпов:

1. **Дорожная карта команды**: Определяет общие инициативы и направления развития продукта. Инициативы делятся на эпики, а каждый эпик содержит несколько requirements и user stories.
2. **Requirements**: Это конкретные задачи, которые команда планирует выполнить для достижения целей продукта. [Владелец продукта упорядочивает их по приоритету](https://www.atlassian.com/ru/agile/scrum/backlogs).

Расстановка приоритетов в backlog зависит от важности для клиента, необходимости обратной связи, относительной сложности реализации и других факторов. [Совместными усилиями команда обеспечивает оптимальную рабочую нагрузку и успешную поставку продукта](https://www.atlassian.com/ru/agile/scrum/backlogs).

**Refinement Meeting**

[**Refinement Meeting** (или **Уточнение Бэклога Продукта**) — это коллаборативное собрание команды, обычно в рамках процесса Agile-разработки программного обеспечения, на котором обсуждаются и уточняются **пользовательские истории** (требования) для предстоящих спринтов или итераций](https://www.atlassian.com/agile/project-management/backlog-refinement-meeting) (итерация — это один шаг циклического процесса). Цель такого собрания — убедиться, что пользовательские истории хорошо определены, правильно оценены и готовы к разработке.

Refinement Meeting - это совещание, которое проходит в рамках Agile-подхода к разработке программного обеспечения и предназначено для обсуждения, уточнения и детализации задач, которые должны быть выполнены в ближайшем будущем. Целью Refinement Meeting является обеспечение понимания всеми членами команды проекта технических и функциональных требований к разрабатываемому продукту, а также определение способов их реализации. В ходе совещания команда отвечает на вопросы, выясняет необходимые детали и оценивает сложность задач, которые могут возникнуть в процессе их реализации. Refinement Meeting помогает предотвратить проблемы и дополнительные затраты на исправление ошибок, путем уточнения требований и улучшения качества работы команды.

**Grooming in Agile Scrum**

[**Grooming** (или **рефайнмент**) в **Agile Scrum** — это важный этап, на котором команда обсуждает детали **бэклога продукта** и готовит его к следующему спринту1](https://habr.com/ru/companies/hygger/articles/354378/). Во время груминга представители **Scrum-команды** обсуждают задачи, которые находятся в бэклоге, чтобы лучше понять их функциональность и необходимые детали. [Это включает в себя аспекты дизайна, интеграции и ожидаемого взаимодействия с пользователем2](https://www.andplus.com/blog/agile-development-process-what-is-grooming). После груминга задачи готовы к передаче разработчикам для начала выполнения.

is a key agile practice and one of the scrum ceremonies for the Scrum team to keep the Product Backlog items refined, up-to-date, prioritized, and ready for the upcoming sprint planning. It is an ongoing activity for adding, refining, estimating, removing, re-ordering, splitting, or merging product backlog items. The key participants of this meeting are the product owner, the Scrum master, and the development team.

# What is a sprint planning meeting?

A sprint planning meeting is a working session for development teams to decide which backlog items to prioritize in the next [sprint](https://www.aha.io/roadmapping/guide/agile/what-is-a-sprint). Attendees include the scrum master, product manager, and members of the [scrum](https://www.aha.io/roadmapping/guide/agile/what-is-scrum) team. Together, they work to set sprint goals, determine how the work will be completed, and align on next steps.

Naturally, sprint planning comes first in the sequence. This is when the team collectively commits to the batch of work for each sprint, which is typically two to four weeks long. You will determine what you will build, test, and release to customers.

 here is a quick overview:

|  |  |
| --- | --- |
| **What is the objective of sprint planning meetings?** | To align on a sprint goal and determine which items from the [backlog](https://www.aha.io/roadmapping/guide/release-management/what-is-a-product-backlog) will be included in the sprint |
| **What are the key agenda items?** | Designating a sprint goal, discussing capacity, reviewing the backlog, and assigning work |
| **Who attends sprint planning?** | Product owners, scrum masters, and the development team |
| **When do sprint planning meetings happen?** | At the start of each sprint, usually every two to four weeks |
| **How long are these meetings?** | Generally, sprint meetings last one to two hours for every week of a sprint. For instance, if your sprint is two weeks long, the sprint planning meeting should take no more than four hours. |

## What is a Sprint Retrospective?

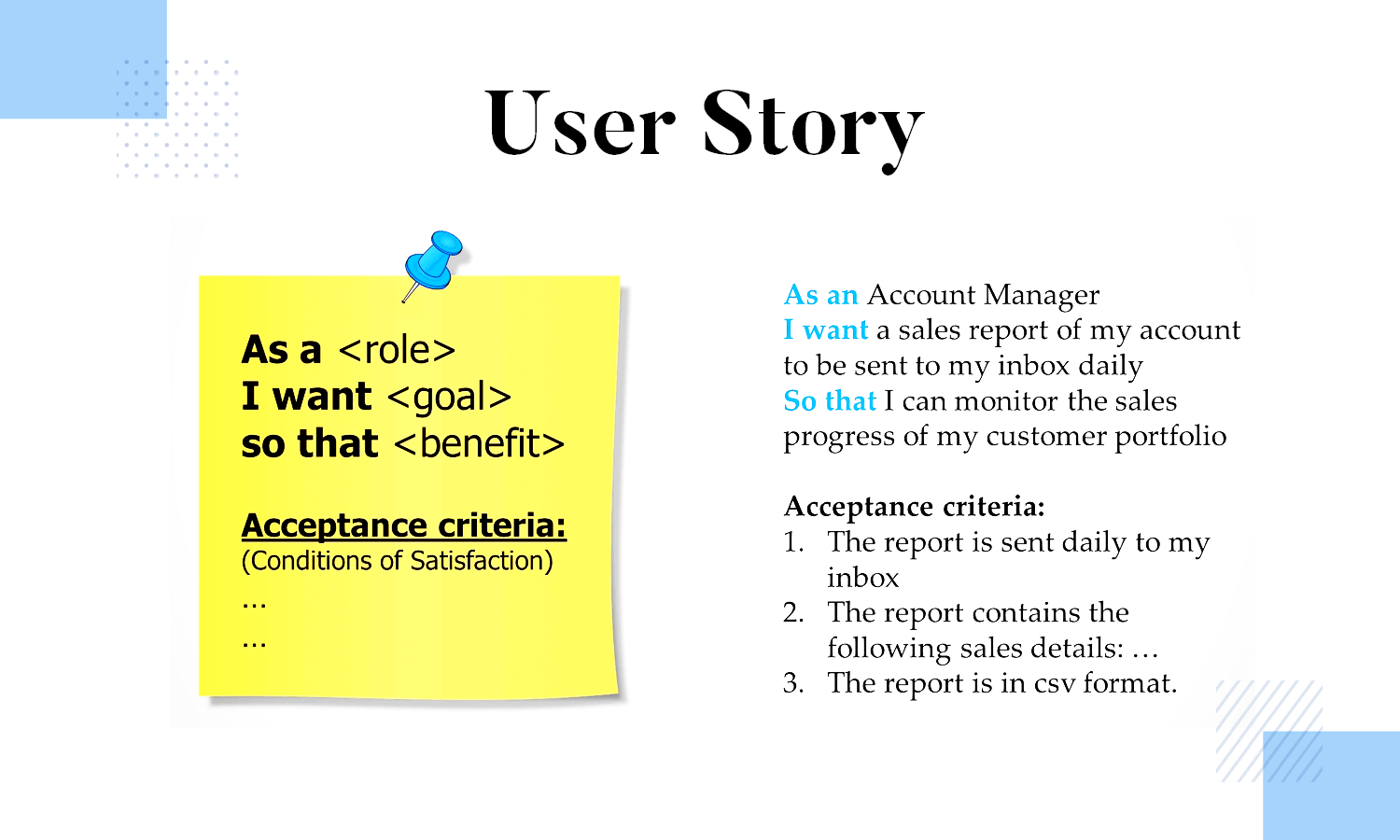
[Sprint retrospectives](https://www.aha.io/roadmapping/guide/agile/what-is-an-agile-retrospective) are held after a sprint concludes — usually immediately after the [sprint review.](https://www.aha.io/roadmapping/guide/agile/sprint-review) During the retrospective, agile development teams discuss what went well, what did not go well, and what you can do to make the next sprint even better.

Takim içi muhasebe. Takımın kendı performansına yönelik muhasebe.

## What is a SPRINT REVIEW?

A sprint review is a collaborative meeting that is typically held **at the end of every sprint**. This is when the team runs through work items they completed during the sprint or iteration. A sprint review ensures key stakeholders are up to date, and it enables them to provide feedback. At the end of each sprint review, the product owner or the team lead uses this feedback to create necessary work items in the product backlog.

**USER STORY:**



**Пользовательская история** — это описание функциональной возможности программного обеспечения, составленное с точки зрения конечного пользователя или клиента. Она пишется простым языком, без технических деталей, и служит контекстом для команды разработчиков. [Пользовательские истории помогают в организации работы, ориентированной на пользователей, и способствуют укреплению сотрудничества, поиску нестандартных идей и повышению качества продукта в целом1](https://www.atlassian.com/ru/agile/project-management/user-stories)[2](https://bing.com/search?q=%D1%87%D1%82%D0%BE+%D1%82%D0%B0%D0%BA%D0%BE%D0%B5+user+story)[3](https://creonit.ru/blog/chto-takoe-user-story-i-kak-ispolzovat-ih-na-proektah/).

Вот небольшой пример пользовательской истории:

***Как покупатель****, я хочу****иметь возможность добавлять товары в корзину****, чтобы****легко совершать покупки онлайн****.*

Эта история описывает потребность пользователя (покупателя) и цель (легкость онлайн-покупок) в простой и понятной форме. [После создания такой истории команда разработчиков может приступить к ее реализации, учитывая потребности клиента](https://www.atlassian.com/ru/agile/project-management/user-stories).

**ACCEPTANCE CRITERIA: (Developer should work/change according to this AC) (Tester should do STATIC TEST on acceptance criteria)**

[**Acceptance Criteria** («критерии приема работы») — это условия, которые позволяют вам определить, когда ваша пользовательская история (User Story) завершена и обладает всеми функциями, необходимыми для удовлетворения потребностей вашего пользователя, вашего клиента](https://leadstartup.ru/agile-coach/acceptance-criteria). Эти критерии помогают уточнить, что должно быть реализовано, чтобы считать задачу выполненной и готовой к принятию.

Важно, чтобы **Acceptance Criteria** были ясными, конкретными и измеримыми. Они помогают команде разработчиков и заказчику понять, что ожидается от реализации функциональности. Например, если вы разрабатываете форму входа на веб-сайт, критерии приема могут включать в себя следующие условия:

1. Пользователь может ввести свой логин и пароль.
2. После успешного входа, пользователь перенаправляется на главную страницу.
3. Если введены неверные учетные данные, пользователь получает сообщение об ошибке.

Такие четкие критерии помогают избежать недоразумений и уточнить ожидания всех заинтересованных сторон.

Acceptance criteria are defined as the conditions that must be satisfied for a product, user story, or increment of work to be accepted. While not a part of the Scrum Guide, AC can be a useful tool teams may choose to use to improve the quality of product backlog items.

Also shortened to the acronym AC, these conditions are pass/fail. Acceptance criteria are either met or not met; they're never only partially fulfilled.

AC are often expressed as a set of statements. These statements should be:

* Clear, so that everyone understands them
* Concise, so that there’s no ambiguity
* Testable or verifiable
* Focused on providing customer-delighting results

Acceptance criteria do not focus on “how” a solution is reached or “how” something is made. Instead, they illuminate the “what” of the work you are doing. For example, the criteria may be:

*Users can pay with Google Pay or Apple Pay at checkout.*

The spirit of acceptance criteria is not to tell you how to do it, for example:

*Install a Wordpress plugin that allows you to create a checkout page.*

Or -

*Write HTML that makes it possible to pay with Apple Pay or Google Pay.*

These statements get at how the work will be done, not the conditions for accepting the work. It’s up to the developers on the scrum team to decide the **how** of fulfilling the acceptance criteria.

The agile practice of formally stated acceptance criteria started in software development, but today they are applied to a wide array of deliverables in addition to software products across diverse industries, from app development to Human Resource departments and beyond.

**! ! ! User story describes the “why” of the work, while the acceptance criteria describe the “what.” The “how” is decided by developers as they work through the sprint.**

## What is a Test Case?

**Тест-кейс** (или **Test Case**) — это пошаговое описание действий, которые нужно произвести для проверки какой-либо функции программного обеспечения (ПО). [Простыми словами, это алгоритм, по которому тестировщик должен пройти (смоделировать поведение пользователя), чтобы проверить работоспособность определенного куска кода](https://testirovshik.com/test-case/).

Вот пример тест-кейса:

1. **Название тест-кейса**: Уведомление пользователя о снижении заряда аккумулятора вручную
2. **Предусловия тест-кейса**: Статус самоката: в аренде
3. **Шаги тест-кейса**:
   * Шаг 1: Зайти на сайт samokat.admin
   * Шаг 2: Логин — test, пароль — test
   * Шаг 3: Перейти на вкладку «Самокаты в аренде»
   * Шаг 4: Нажать…
   * Шаг 5: Включить…
4. [**Ожидаемый результат тест-кейса**: Появляется сообщение об успешном выполнении тест-кейса «Пользователь уведомлен о снижении заряда»](https://solvery.io/blog/ru/interesting/chto-takoe-test-case-primer-i-check-list-dlya-nachinayushhih-testirovshhikov/).

Правильно составленные тест-кейсы помогают команде проверить программу и познакомиться с ней, не читая весь код, а изучив только текстовое описание тест-кейса. [Важно, чтобы тест-кейсы были четко сформулированы и содержали всю необходимую информацию для проведения тестирования](https://solvery.io/blog/ru/interesting/chto-takoe-test-case-primer-i-check-list-dlya-nachinayushhih-testirovshhikov/).

Test cases are specific steps in software testing to verify features, including data, prerequisites, and postconditions for verification.

A test case serves as a predefined set of conditions or activities designed to validate the functionality of a software program or system. It encompasses a comprehensive description of the inputs, preconditions, steps to be executed, and expected outcomes. Test cases are meticulously written to target specific aspects or functionalities of the software, ensuring that they align with the established requirements and meet the desired objectives.

The primary purpose of running test cases is to identify any defects or issues in the application so that they can be fixed before the application is released to the end users.

Here are some key points that will give you a good understanding of test cases:

**Manual and Automated Test Case Creation:** You can create testing cases manually or by automated approach.

**Manual Test Case Creation and Execution:** The tester creates manual testing cases and follows the steps outlined to verify the application’s functionality.

**Automated Test Case Execution with Test Tools:** Using [test tools](https://www.lambdatest.com/learning-hub/test-tool) and frameworks, automated testing cases are executed, which follow Software Requirement Specification (SRS).

**Structured Approach to Functionality Verification:** It provides a structured approach to verify the functionality of software applications.

**Independence of Test Cases:** They are not dependent on each other, meaning that the result of one test case should not impact the result of another.

**Controlled Execution Environment for Test Cases:** You can execute the test case in a controlled environment, ensuring the availability of all necessary resources without impacting the software production environment.

|  |  |
| --- | --- |
| ****Gherkin Language Example:**** Scenario: Add two numbers  Given I have entered 50 into the calculator  And I have entered 70 into the calculator  When I press add  **Then** the result should be 120 on the screen | Below is an example to write a test case for Gherkin for the login form:  Scenario: Login as Admin  Given I Navigate to the login page  And I Enter username and password  When I click to login button  **Then** I can see admin is able to login |
| **Let’s see one feature file where we use all the keywords:**  Feature: User Login  As a registered user,  I want to log in with my username and password,  So that I can access my account dashboard.  Background: User is on the Login Page  Given the user has navigated to the login page  Scenario: Successful Login with Valid Credentials  Given the user has entered a valid username and password  When the user clicks on the login button  Then the user should be redirected to the dashboard  Scenario: Unsuccessful Login with Invalid Credentials  Given the user has entered an invalid username or password  When the user clicks on the login button  Then the user should see an error message | |

TASK / EPIC / TICKET

Task – aldığımiz iş/görevler mecmuası = Ticket işin yazıya dökülmesi

Epic – ana senaryo

В **Agile** методологии, такой как **Scrum** или **Kanban**, используются различные термины для организации работы. Давайте рассмотрим их:

1. **Эпик (Epic)**:
   * **Эпик** — это большой объем работы, который можно разбить на несколько историй поменьше. В **Jira** (популярной системе управления проектами) такие истории называются **“задачами”**.
   * **Эпики** часто объединяют несколько команд, работающих над разными проектами. Они могут отслеживаться на разных досках.
   * Обычно **эпик** выполняется за несколько спринтов. [По мере получения командой больше информации об эпике, набор пользовательских историй в нем может меняться](https://www.atlassian.com/ru/agile/project-management/epics)[1](https://www.atlassian.com/ru/agile/project-management/epics).
   * Пример эпика: представьте, что мы работаем в турфирме, организующей космические полеты. [Эпик “Запуск туристического космического корабля, март 2050 г.” включает истории, связанные с покупкой билетов на полет и улучшением ключевых аспектов запуска ракеты](https://www.atlassian.com/ru/agile/project-management/epics)[1](https://www.atlassian.com/ru/agile/project-management/epics).
2. **Задача (Task)**:
   * **Задача** — это конкретная единица работы, которая может быть выполнена отдельным исполнителем. В **Jira** они также называются **“задачами”**.
   * **Tasks** обычно более мелкие по объему, чем **epics**, и могут быть выполнены за один спринт.
   * Пример задачи: написать код для определенной функциональности в приложении.
3. **Тикет (Ticket)**:
   * **Тикет** — это общий термин, который может включать в себя как задачи, так и эпики. Он используется для отслеживания работы и управления проектами.
   * [В **Jira** тикеты могут быть разных типов, таких как **epics**, **tasks**, **bags etc.**](https://wbtech.ru/jira/jira-issue-types/)

Важно понимать, что эти термины могут немного различаться в разных методологиях и инструментах управления проектами, но общая идея остается примерно такой же.

### Description

Edit

### User Story

* **Nedir?** Bir özelliği kullanıcının bakış açısından anlatan kısa bir ifade.
* **Örnek:** "Bir alışverişçi olarak, seçtiğim ürünleri sepete ekleyebilmek istiyorum, böylece satın alma işlemi yapabilirim."

### Acceptance Criteria

* **Nedir?** User story'nin başarıyla tamamlandığını gösteren ölçütler.
* **Örnek:**
  1. Kullanıcı "Sepete Ekle" butonuna tıklayarak ürünü sepete ekleyebilir.
  2. Sepetin içeriği güncellendiğinde kullanıcıya bildirim gösterilir.

### Test Case

* **Nedir?** Belirli bir özelliği test etmek için adım adım takip edilen yönergelerin tamami.
* **Örnek:**
  1. Ürün sayfasını aç.
  2. "Sepete Ekle" butonuna tıkla.
  3. Ürünün sepete eklendiğini ve bildirimin göründüğünü kontrol et.

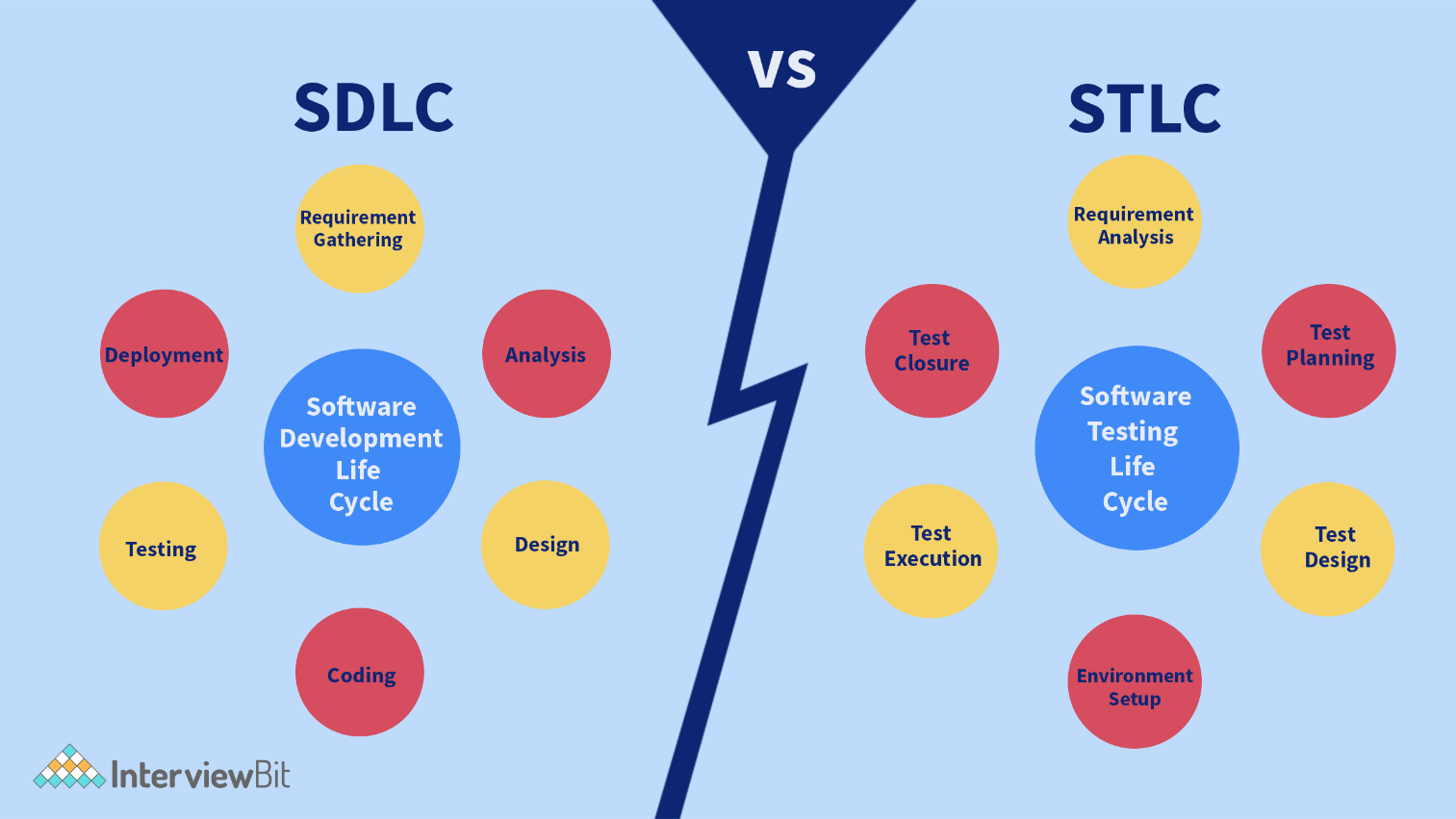
### Epic ve Task

* **Epic:** Birden fazla user story'yi kapsayan büyük bir hedef.
  + **Örnek:** "Alışveriş Sepeti İşlevselliğinin Geliştirilmesi" epic'i, sepete ekleme, çıkarma gibi birden fazla user story içerebilir.
* **Task:** User story veya epic'in tamamlanması için gerekli küçük görevler.
  + **Örnek:** "Sepete Ekle Butonunun Kodlanması" bir task olabilir.

### Farklar ve İlişkiler

* **User Story vs Acceptance Criteria:** User story, "ne" yapılacağını anlatır. Acceptance criteria ise bu story'nin "nasıl" başarılı sayılacağını belirtir.
* **Epic vs User Story:** Epics, genelde büyük hedeflerdir ve birden fazla user story içerir. User story'ler ise daha spesifik ve yönetilebilir özelliklerdir.
* **Task:** Hem epics hem de user story'ler, tamamlanması gereken spesifik görevlere (task) bölünebilir. Bu task'lar, o story'nin veya epic'in gerçekleştirilmesi için atılması gereken adımlardır.

Soft Skils Day 4

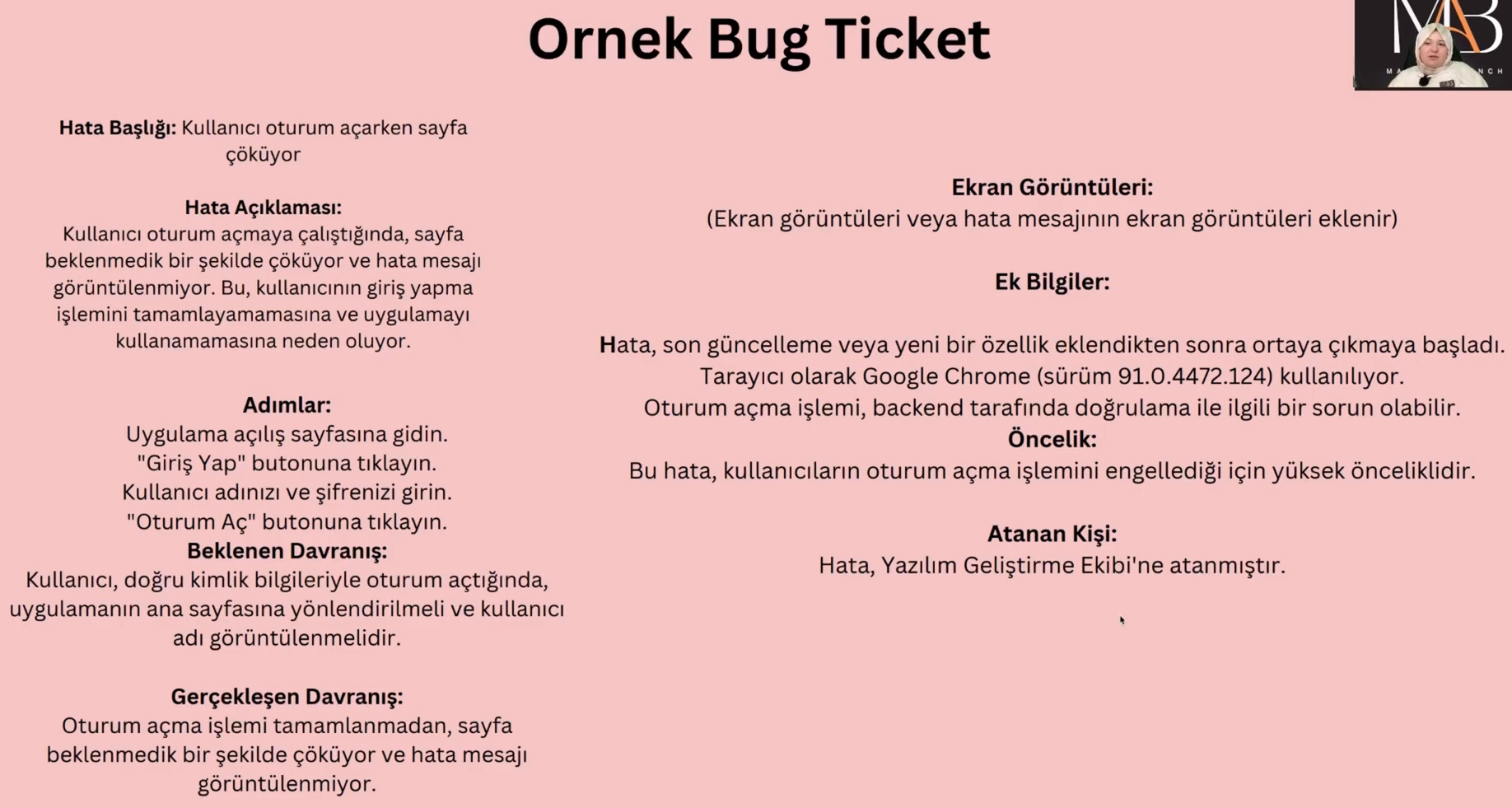


# Difference between SDLC and STLC

|  |  |
| --- | --- |
| [Software Development Life Cycle (SDLC)](https://www.geeksforgeeks.org/problems/software-development-life-cycle) is a sequence of different activities performed during the software development process. | [Software Testing Life Cycle (STLC)](https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/) is a sequence of different activities performed during the software testing process. |

**Difference between SDLC and STLC:**

| **SDLC** | **STLC** |
| --- | --- |
| SDLC is mainly related to software development. | STLC is mainly related to software testing. |
| Besides development other phases like testing is also included. | It focuses only on testing the software. |
| SDLC involves total six phases or steps. | STLC involves only five phases or steps. |
| In SDLC, more number of members (developers) are required for the whole process. | In STLC, less number of members (testers) are needed. |
| In SDLC, development team makes the plans and designs based on the requirements. | In STLC, testing team(Test Lead or Test Architect) makes the plans and designs. |
| Goal of SDLC is to complete successful development of software. | Goal of STLC is to complete successful testing of software. |
| It helps in developing good quality software. | It helps in making the software defects free. |
| SDLC phases are completed before the STLC phases. | STLC phases are performed after SDLC phases. |
| Post deployment support , enhancement, and update are to be included if necessary. | Regression tests are run by QA team to check deployed maintenance code and maintains test cases and automated scripts. |
| Creation of reusable software systems is the end result of SDLC. | A tested software system is the end result of STLC. |



**Day 5**

### ****Defect (дефект)****

Дефект — это ситуация, при которой приложение не работает соответственно требованиям, то есть отличается ожидаемое и действительное поведение продукта.

**Defect**: A defect occurs when the application does not work as per the specified requirements. The actual and expected results of the software are not in sync. Defects can arise due to mistakes made by developers during the development phase.

### ****Error (ошибка)****

Error — это ситуация, которая происходит когда разработчики неправильно понимают требования к продукту, и это приводит к багам.

**Error**: An error happens when the development team or developer misunderstands a requirement definition, leading to buggy code. Errors can result from wrong logic, syntax, or loops, impacting the end-user experience.

### ****Bug (баг)****

Баг — это ситуация, возникающая, когда продукт (приложение) не соответствует утвержденным ранее требованиям. Если где-то в коде допущена ошибка, она провоцирует «поломку» кода, и приложение работает некорректно. Такую ситуацию тестировщики называют багом. Найденный баг может быть воспроизведен, то есть воссозданы условия, при которых он возникает.

**Bug**: A bug refers to a **defect** in the software product or application. It means that the software is not working as per the adhered requirements. Bugs can result from logical errors, causing the code to break. Automation or manual test engineers identify these situations as bugs. Common types of bugs include crashes and typos.

### ****Fault (сбой)****

Ситуация, когда приложение не может правильно функционировать. Например, из-за недостатка свободной памяти или места на диске, или невыполнения каких-то нужных действий.

**Fault**: A fault occurs when the software fails due to factors like lack of resources or not following proper steps. It represents unintended behavior by an application program.

### ****Failure (отказ)****

Сочетание дефектов, которые приводят к полному отказу приложения, обычно с потерей данных, в том числе критически важных; система не реагирует на действия пользователя. Обычно такие ситуации происходят редко — перед релизом продукта тестируются возможные сценарии.

Failure in software refers to a situation where the software does not perform its intended function or produces incorrect results.

## Таблица: Bug — Defect — Error — Fault — Failure

|  | **Bug (баг)** | **Defect (дефект)** | **Fault (сбой)** | | **Error (ошибка)** | **Failure (отказ)** | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Описание проблемы | Продукт не работает как должен работать соответственно требованиям | Несоответствие полученного и ожидаемого результата | Состояние приложения, при котором оно не выполняет свою функцию | | Ошибка, допущенная в коде, при компиляции или выполнении, из-за которой происходит Fault | Серьезный дефект или их сочетание, приводящее к полному отказу приложения, потере данных и потере управления приложением | |
| Кто сталкивается | Тестировщики | Находят тестировщики и устраняют разработчики | Пользователи | | Тестировщики и разработчики | Тестировщики и пользователи | |
| Какие типы бывают | Много типов, чаще всего: — Логические баги — Алгоритмические баги — Баги распределения системных ресурсов | — Высокий, средний и низкий приоритет — Критическая, большая, малая и очень низкая серьезность | — Бизнес-логики — Функциональные — Интерфейса — Производительности — Аппаратные | | — Синтаксические ошибки — Ошибки в интерфейсе — Ошибки рантайма — Ошибки вычислений — Аппаратные ошибки | Неработоспособность приложения, по разным причинам | |
| Причины возникновения | — Ошибки в логике приложения — Дефектный код | — Получение и передача некорректных данных, из-за некорректной логики/архитектуры/другие причины | — Неправильная архитектура приложения — Неправильная логика приложения | — Ошибки в коде — Ошибки в логике — Неспособность приложения к компиляции и выполнению — Неправильно понятые требования — Неправильная архитектура приложения | | | — Некорректные переменные окружения — Системные ошибки — Человеческие ошибки |
| Как предотвратить | — Разработка через тестирование (TDD) — И другие современные практики разработки и тестирования, стимулирующие «чистоту кода» | — Внедрение современных практик разработки — Правильное применение существующих практик | — Проверка кода и требований другими участниками команды — Тщательная верификация архитектуры и кода | | — Проверка кода другими участниками команды — Валидация багфиксов | — Повторное тестирование — Тщательное сквозное тестирование — Проверка требований/спецификаций — Категоризация ошибок и проблем | |

The common reasons behind any software failure are:

* False architecture definition and low level design.
* A forced schedule or milestone dates without substantial amount of data and analysis.
* Failing to account and adjust for requirement growth.
* Integrating excessive personnel to achieve unrealistic schedule compression.
* Intuition-based or emotional stakeholder negotiation.
* Miscommunication, egos and negative attitudes.

**Day\_6**

**Black box testing** (тестирование черного ящика) - это методика тестирования программного обеспечения, при которой **тестировщик не знает** внутренних механизмов или структуры кода системы. Другими словами, при таком тестировании фокус делается исключительно на **внешнем поведении** программы, без доступа к ее внутреннему исходному коду. Название “черный ящик” происходит от идеи, что внутренние механизмы скрыты или “запечатаны” от взгляда тестировщика.

Основные характеристики тестирования черного ящика включают:

1. **Независимое тестирование**: Тестирование черного ящика обычно выполняется тестировщиками, независимыми от команды разработчиков. Это обеспечивает свежий взгляд и помогает выявить проблемы, которые могли бы ускользнуть от разработчиков.
2. **Тестирование на основе требований**: Тестировщики разрабатывают тестовые случаи на основе требований и спецификаций программного обеспечения, не беспокоясь о том, как код реализован.
3. **Функциональное тестирование**: Основная цель тестирования черного ящика - оценить функциональность программы, проверить, соответствует ли она ожидаемому поведению и предоставляет ли желаемые результаты для разных входных данных.
4. **Отсутствие знания внутреннего кода**: Тестировщики не имеют доступа к исходному коду, архитектуре или деталям проектирования программы. Они взаимодействуют с системой через ее пользовательский интерфейс или API.

Тестирование черного ящика включает в себя несколько типов тестирования, каждый из которых имеет свою специфику и цель. [Этот метод помогает обнаруживать проблемы с точки зрения пользователя и повышает надежность, пользовательский опыт и общее качество программного обеспечения](https://www.browserstack.com/guide/black-box-testing).

**White box testing** (или **тестирование структурное**) — это подход к тестированию программного обеспечения, при котором тестировщик имеет доступ к **внутренним механизмам и коду** системы. В отличие от тестирования черного ящика, где фокус на внешнем поведении, тестирование белого ящика учитывает **внутренние детали**.

Основные характеристики тестирования белого ящика:

1. **Структурное тестирование**: Тестировщик анализирует **структуру кода**, проверяет ветви, операторы, условия и пути выполнения.
2. **Покрытие кода**: Оценивается, насколько код был протестирован. Это включает покрытие операторов, условий, путей и функций.
3. **Интеграция с разработкой**: Тестирование белого ящика часто выполняется **на уровне модулей**, но также используется для интеграционного и системного тестирования.

[Этот метод помогает выявить дефекты, связанные с **логикой программы**, и обеспечивает более полное покрытие кода.](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B1%D0%B5%D0%BB%D0%BE%D0%B3%D0%BE_%D1%8F%D1%89%D0%B8%D0%BA%D0%B0)

[**Unit testing** (bu testi developer yapar) — это метод тестирования программного обеспечения, при котором **отдельные модули и компоненты исходного кода** (например, функции, методы, классы) проверяются на соответствие заданным требованиям и корректность поведения1](https://en.wikipedia.org/wiki/Unit_testing). Этот процесс является стандартным шагом в разработке и реализации программ, таких как Agile.

Вот некоторые ключевые аспекты юнит-тестирования:

* **Цель**: Проверить, что отдельные части программы работают правильно и соответствуют ожиданиям.
* **Что тестируется**: Юнит-тесты проверяют функциональность **наименьших изолированных компонентов** приложения.
* **Автоматизация**: Юнит-тесты могут быть автоматизированы и выполняются разработчиками перед более обширным тестированием.
* **Критерии успешности**: Тесты могут содержать критерии, которые должны выполняться для успешного прохождения.
* **Примеры**: Параметризованные тесты, тесты на методы-заглушки, мок-объекты и другие инструменты помогают тестировать модули в изоляции.

Таким образом, юнит-тестирование помогает обнаруживать ошибки на ранних этапах разработки и обеспечивает надежность кода.

**\* \* \***

**Integration Testing** — это метод проверки программного обеспечения, при котором **логически объединенные модули** тестируются как группа. [Основной целью интеграционного тестирования является выявление дефектов во взаимодействии между этими модулями](https://www.guru99.com/integration-testing.html).

Вот некоторые ключевые аспекты интеграционного тестирования:

1. **Цель**: Проверить, что программные модули взаимодействуют правильно и соответствуют ожиданиям.
2. **Что тестируется**: Интеграционное тестирование фоку сируется на проверке **передачи данных** между модулями.
3. **Методы**: Существует несколько подходов к интеграционному тестированию:
   * **Big Bang Testing**: Все компоненты интегрируются сразу и тестируются.
   * **Incremental Testing**: Модули интегрируются постепенно, и каждый этап тестируется.
   * **Stubs и Drivers**: Заглушки и драйверы используются для имитации недостающих компонентов.
   * **Bottom-up Integration Testing**: Тестирование начинается с низкоуровневых модулей.
   * **Top-down Integration Testing**: Тестирование начинается с высокоуровневых модулей.
   * **Sandwich Testing**: Комбинирует подходы сверху вниз и снизу вверх.

Пример интеграционного тест-кейса:

Предположим, у нас есть приложение с тремя модулями: **“Страница входа”**, **“Почтовый ящик”** и **“Удаление писем”**. Нам нужно проверить, как эти модули взаимодействуют друг с другом. Важно обратить внимание на связи между модулями, а не на отдельные функции, которые уже были протестированы. Вот пример тест-кейсов:

| **ID тест-кейса** | **Цель теста** | **Описание теста** | **Ожидаемый результат** |
| --- | --- | --- | --- |
| 1 | Проверка связи между “Страницей входа” и “Почтовым ящиком” | Ввести учетные данные и нажать кнопку входа | Переход к почтовому ящику |
| 2 | Проверка связи между “Почтовым ящиком” и “Удалением писем” | Выбрать письмо в почтовом ящике и нажать кнопку удаления | Выбранное письмо должно появиться в папке “Удаленные” |

Интеграционное тестирование помогает выявить проблемы во взаимодействии между модулями и обеспечивает надежность программного кода.

**\* \* \***

**Acceptance Testing** (тестирование на приемку) — это процесс тестирования черного ящика, в котором проверяется функциональность программного продукта, чтобы убедиться, что он соответствует критериям приемки. [Это последний этап процесса тестирования программного обеспечения, и он важен перед тем, как программное обеспечение становится доступным для реального использования](https://www.browserstack.com/guide/acceptance-testing)[1](https://www.browserstack.com/guide/acceptance-testing).

Важность тестирования на приемку огромна, и если команда тестирования пропустит этот этап, есть риск того, что программное обеспечение не будет соответствовать начальным требованиям. После успешного завершения системного тестирования в жизненном цикле разработки программного обеспечения тестировщики должны провести тест на приемку:

1. **Анализ требований**.
2. **Создание плана тестирования**.
3. **Проектирование тестовых случаев**.
4. **Выполнение тестовых случаев**.
5. **Подтверждение целей**.

Тестирование на приемку выполняется с точки зрения пользователя, чтобы убедиться, что программное обеспечение работает согласно определенным спецификациям. [Оно также помогает подтвердить, что программное обеспечение соответствует текущим стандартам рынка и конкурирует с аналогичными продуктами в отрасли1](https://www.browserstack.com/guide/acceptance-testing).

**\* \* \***

**Smoke Testing** (тестирование на дым) — это процесс тестирования программного обеспечения, который проверяет стабильность и функциональность развернутой сборки программы. Он выполняется после интеграции новых функций с существующей сборкой и перед ее выпуском в тестовую среду (QA). Давайте рассмотрим подробнее:

1. **Цель дымового тестирования**:
   * Убедиться, что критические функции программы работают корректно.
   * Проверить, что нет критических ошибок, которые могли бы привести к отклонению сборки.
   * Это мини-регрессионное тестирование основных функций.
2. **Когда выполняется дымовое тестирование**:
   * После интеграции новых функций с существующей сборкой.
   * Перед выпуском сборки в тестовую среду.
   * Если сборка проходит дымовое тестирование, она допускается к дальнейшему формальному тестированию.
3. **Кто выполняет дымовое тестирование**:
   * QA инженеры или руководители QA.
   * Они проверяют наличие критических ошибок в приложении.
4. **Примеры дымового тестирования**:
   * Проверка основных функций, которые пользователь использует.
   * Проверка поддерживающих функций (например, входа в аккаунт).
   * Проверка навигации и возможности чтения/записи данных.
5. **Зачем нужно дымовое тестирование**:
   * Обеспечить стабильность системы.
   * Выявить крупные проблемы на ранних этапах разработки.

[Таким образом, дымовое тестирование помогает определить, готова ли сборка к дальнейшему тестированию и релизу1](https://testengineer.ru/chto-takoe-smok-testirovanie/)[2](https://en.wikipedia.org/wiki/Smoke_testing_%28software%29)[3](https://www.guru99.com/smoke-testing.html).

**\* \* \***

**Regression Testing** — это важный этап в жизненном цикле разработки программного обеспечения. Давайте разберемся подробнее:

1. **Цель регрессионного тестирования**:
   * Проверить, что изменения в коде не привели к появлению новых ошибок или не активировали ранее необнаруженные проблемы.
   * Гарантировать, что ранее разработанное и протестированное программное обеспечение продолжает работать корректно после внесенных функциональных или нефункциональных изменений.
   * Термин “регрессия” означает “возвращение к более раннему или менее развитому состоянию”.
2. **Зачем важно регрессионное тестирование**:
   * **Высокое качество программного продукта**.
   * **Быстрое обнаружение ошибок**.

**\* \* \***

**Performance testing (tестирование производительности)** — это процесс проверки, который фокусируется на том, как система выполняет свою работу под определенной нагрузкой. Важно отметить, что это не связано с поиском ошибок в программном обеспечении. Вместо этого разные типы тестирования производительности измеряются согласно стандартам и бенчмаркам. Давайте рассмотрим подробнее:

1. **Типы тестирования производительности**:
   * **Тестирование нагрузки (Load Testing)**: Измеряет производительность системы при увеличении нагрузки. Эта нагрузка может быть связана с одновременными пользователями или транзакциями. Система мониторится для измерения времени ответа и устойчивости при увеличении нагрузки.
   * **Тестирование стресса (Stress Testing)**: В отличие от тестирования нагрузки, тестирование стресса (или устойчивости) проводится за пределами нормальных рабочих условий. Программное обеспечение подвергается большему количеству пользователей или транзакций, чем оно может обработать. Цель — измерить стабильность программы и ее восстановление после сбоя.
   * **Тестирование пиковой нагрузки (Spike Testing)**: Это разновидность тестирования стресса, при которой оценивается производительность программы при быстром и повторном увеличении нагрузки. Нагрузка превышает ожидания в течение короткого времени.
   * **Тестирование выносливости (Endurance Testing)**: Оценивает, как программное обеспечение работает с нормальной нагрузкой в течение продолжительного времени. Цель — выявить проблемы, такие как утечки памяти.
   * **Тестирование масштабируемости (Scalability Testing)**: Определяет, эффективно ли программное обеспечение обрабатывает увеличивающуюся нагрузку. Это может быть достигнуто путем постепенного увеличения нагрузки или изменения ресурсов, таких как процессоры и память.
   * **Тестирование объема данных (Volume Testing)**: Определяет, насколько эффективно программное обеспечение работает с большим объемом данных. Также известно как тестирование на “потопление”, так как система затопляется данными.

**\* \* \***

**Day\_7**

**Git** and **GitHub** are fundamental tools in the world of software development.

1. **Git**:

**Definition**: Git is a **version control system** (VCS) that allows developers to manage and keep track of changes in their source code over time.

**How It Works**: When you’re working on a project, Git helps you record snapshots of your code at different points. These snapshots are called **commits**. You can think of Git as a way to document the evolution of your code.

**Key Features**:

**Distributed**: Git allows you to work on your code from different computers and collaborate with other developers. It’s not tied to a central server.

**Local**: You can make commits and track changes locally on your machine.

**Branching**: Git enables you to create branches to work on specific features or fixes without affecting the main codebase.

**Merging**: You can merge changes from one branch into another.

**Creator**: Linus Torvalds developed Git in 2005 as open-source software.

**Use Case**: Git is essential for individual developers and teams working on projects.

1. **GitHub**:

**Definition**: GitHub is a **cloud-based hosting service** that provides a platform for managing Git repositories.

**How It Works**: Developers use GitHub to host, share, and collaborate on their code. It’s like a social network for developers, where they can contribute to open-source projects, report issues, and collaborate.

**Key Features**:

**Repository Hosting**: GitHub hosts Git repositories, making it easy to share code with others.

**Collaboration**: Multiple developers can work together on the same project, contributing code and managing issues.

**Pull Requests**: Developers can propose changes (pull requests) to a project, which are reviewed and merged if accepted.

**Issues and Discussions**: GitHub provides tools for tracking issues, discussing features, and managing project tasks.

**Maintainer**: Microsoft maintains GitHub.

**Use Case**: GitHub is widely used for open-source projects, team collaboration, and code sharing.

In summary, **Git** is the version control system that manages your code, while **GitHub** is the platform where you host and collaborate on that code. [Learning Git and GitHub is crucial for anyone entering the world of software development, as they are essential tools for effective collaboration and code management](https://www.freecodecamp.org/news/introduction-to-git-and-github/).

1. [**Sprint** в разработке программного обеспечения — это ограниченный по времени период, в течение которого команда Scrum выполняет определенное количество задач с целью достижения четко определенной цели](https://www.softwaretestinghelp.com/versionone-tutorial/).

   Вот некоторые ключевые аспекты спринта:

   **Длительность**: Спринты обычно планируются на примерно **две недели**, хотя они могут быть короче (неделя) или длиннее (месяц).

   **Инкрементальные изменения**: Важное преимущество короткого временного интервала спринта заключается в том, что разработчики фокусируются на выпуске **маленьких, пошаговых изменений**, а не крупных и радикальных. Это уменьшает объем отладки и позволяет пользователям получать более плавный опыт работы с продуктом.

   **Типы задач**: В спринте можно выполнять различные задачи, включая **фронтенд-разработку**, **бэкенд-разработку**, **управление базой данных** и даже **машинное обучение**.

   **Постепенные изменения**: Спринты позволяют внедрять изменения **постепенно**, что помогает пользователям привыкнуть к новым функциям без резких скачков.

   **Зачем используют спринты**:

   **Сокращение времени отладки**: Меньший объем кода в спринте упрощает процесс отладки, поскольку меньше нужно проверять. Это как игра в “Где Волдо?” — проще найти Волдо на открытке размером с открытку, чем на плакате размером с постер.

   **Постепенное внедрение изменений**: Пользователи легче привыкают к постепенным изменениям, чем к резким. Спринты позволяют внедрять новые функции без драматических изменений.

   [Таким образом, спринты обеспечивают предсказуемость, быстрое обучение и регулярный ритм разработки](https://www.softwaretestinghelp.com/versionone-tutorial/). [↑](#footnote-ref-1)