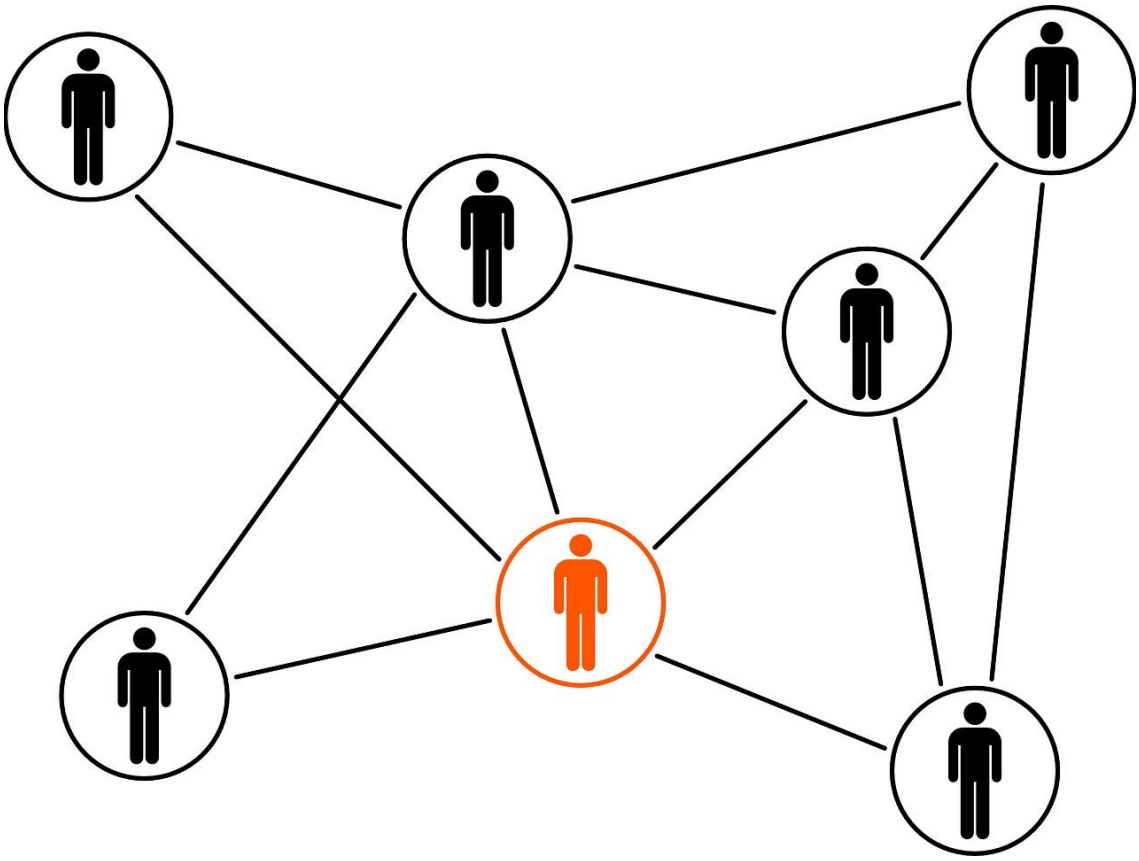
# Тім лідер (командний лідер) — теоретичний мінімум

Рік: 2024;

Автор: Дмитро Попов (Alma mater ЧНУ), Чернівці. https://www.linkedin.com/in/dima1popov/
Працюю розробником більше 8 років, був технічним лідером на декількох проєктах (перший досвід тех лідера отримав в 2016). Сертифікований Скрам розробник.

Засновано на досвіді автора та Professional Agile Leadership™ Certification (PAL I), також на SWECOM (IEEE) і PMBOK Guide (PMI).

Залучені матеріали з книг:
1. Фред Брукс, "Міфічний людино-місяць" (1975),
2. Геральд Вайнберг, "Стати технічним лідером" (1986),
3. Кент Бек, "Екстремальне програмування" (1999),
4. Генк Рейнвотер, "Як пасти котів" (2002),
5. Роберт Мартін, "Гнучка розробка" (2002),
6. Стівен Палмер, "Практичний посібник з функціонально-орієнтованої розробки" (2002),
7. Майк Коєн, "Гнучке оцінювання та планування" (2005),
8. Кен Швабер та Джефф Сазерленд, "Посібник зі Скраму" (2010).



# Зміст

# Хто такий технічний лідер в команді?

"Лідерство — це не про знання правильної відповіді, а про вміння знаходити правильну відповідь"
(Кріс Латтнер, розробник мови Swift)

Перед розглядом посади "технічний лідер", розглянемо структуру команди.

В книзі "Міфічний людино-місяць" Фред Брукс пише, що команди розробників, які спільно працюють над задачею, повинні бути невеликі, приблизно 10 людей.
Джефф Сазерленд і його Scrum методологія також каже, що команда повинна складатися з 4-10 людей.
Якщо проєкт величезний, то організовуються декілька самодостатніх команд, кожна з яких працює над своєю частиною проєкту. Це може навіть відбиватися на архітектурі проєкту, зокрема, використовують мікросервісну архітектуру та мікрофронтенд. Між-командну взаємодію виконують лідери команд та технічний директор (CTO).

І Фред Брукс і Scrum-методологія вважають, що команда повинна бути крос-функціональна.

Команда з крос-функціональним складом включає в себе фахівців із різних областей, необхідних для виконання проєкту чи завдання. У контексті Scrum це зазвичай означає наявність учасників із різними навичками у сферах дизайну, розробки, тестування та інших необхідних компетенцій. Це різноманіття допомагає команді вирішувати різні аспекти проєкту без постійного залучення зовнішніх ресурсів, сприяючи ефективності та співпраці. QA спеціаліст (тестувальник) вважається частиною Scrum команди та слугує, щонайменше для двох цілей:
1. Програміст не повинен виключно сам тестувати свій код (як каже книга "Мистецтво тестування" Гленфорда Майерса).
2. Звіт менеджеру проєкту про відсутність виявлених багів (помилок). Помилкою (багом) вважається відхилення від специфікації.

Відповідно до Scrum, команди є **багатофункціональними** (cross-functional), тобто учасники мають усі навички, необхідні для створення цінності кожного спринту (ітерації розробки). Хоча багатофункціональні команди повинні мати різноманітний набір навичок, це не означає, що кожен член повинен бути експертом у всіх сферах. Основна увага приділяється володінню навичками, необхідними для поступового постачання продукту. У той час як команди можуть вибрати людей, які мають Т-подібний набір знань, деякі команди можуть вибрати експертів з бізнес-аналізу або експертів з тестування.  (Scrum is defined in the Scrum Guide by Ken Schwaber and Jeff Sutherland).

**Т-подібні навички** (T-shaped skills) — це комбінація глибоких знань у конкретній області (вертикальна частина "Т") і широкого спектра загальних навичок (горизонтальна частина "Т"). Це означає, що людина має глибокі експертні знання в одній області, але також може працювати та спілкуватися ефективно в інших галузях.

Технічний лідер та лідер команди повинні мати Т-подібні навички, а не фреймове мислення.

Технічний лідер відповідальний за технічну придатність проєкту. Він повинен контролювати, щоб інші, менш досвідчені розробники, не зробили критичні помилки.

Фред Брукс пропонує розглядати технічного лідера в команді, як хірурга в операційній. В операційній є головний хірург і його асистенти, які йому допомагають. Хірург робить головні рішення. В такому випадку маємо чітку ієрархію в команді та обов'язки. Звісно, що така команда не може бути великою, до 10 людей. Цей підхід особливо добре працює, якщо в команді є початківці, або спеціалісти середнього рівня, які не мають достатньої експертизи. Цей підхід працює навіть, якщо в команді всі спеціалісти високого рівня. Тех лід пояснює структуру проєкту й каже, що кому робити відповідно до його навичок. Тех лід може консультуватися з учасниками команди, впроваджувати перехресне рев'ю коду. На тех ліді відповідальність за технічну частину проєкту, тому він повинен контролювати дії та

знання інших розробників, вказувати на те, що потрібно вивчити й змінити.
Важливо зауважити, що члени команди не повинні боятися бути ініціативними й навіть показувати, що можливо є краще рішення, ніж те, що пропонує технічний лідер. Але вони ні в якому разі не повинні наполягати на ньому, бо відповідальний саме технічний лідер за критичні рішення, а інші члени команди відповідальні за свій спектр задач і за те, як вони комунікують з лідером.

Технічний лідер може виконувати роль тім лідера, але також це можуть бути окремі ролі.

**Обов'язки лідера команди** (team lead):
1. Спілкування з Product owner (Власник Продукту (Product Owner) — представляє зацікавлені сторони (stakeholders) та є голосом клієнта).
2. Спілкування з менеджером проєкту.
3. Спілкування з іншими командами.
4. Контроль комунікації в команді.
5. Розгляд кар'єрного росту (dev plan).
6. Пріоритет задач.
7. Складання roadmap та беклогу (списку задач).
8. Комунікація з технічним лідером.
9. Деякі технічні задачі, код рев'ю.
10. Документація.

Тім лідер може бути Скрам майстром, а проєктний менеджер мати роль Продукт овнера.

У чудовій книзі про технічне лідерство Джеррі Вайнберг пропонує **модель лідерства MOI**:
**М**отивація: Здатність заохочувати технічних людей робити якнайкраще.
**О**рганізація: Здатність формувати існуючі процеси (або винаходити нові), які дозволять втілити початкову концепцію в кінцевий продукт.
**І**деї чи інновації: Здатність заохочувати людей творити та відчувати себе творчими, навіть якщо вони повинні працювати в межах, встановлених для певного програмного продукту чи програми.

Вайнберг пропонує успішним керівникам проєктів застосовувати стиль управління, орієнтований на вирішення проблем. Тобто керівник проєкту програмного забезпечення повинен зосередитися на розумінні проблеми, яку потрібно вирішити, керувати потоком ідей і водночас повідомляти всім членам команди (словами та, що набагато важливіше, діями), що якість має значення і що це не буде скомпрометовано.

*Team lead (тім лід) тільки один в команді*, технічних лідерів може бути декілька. Наприклад, командний лідер може контролювати всю Scrum крос-функціональну команду, в якій є група бекенд розробників та група фронтенд розробників, кожна з яких має свого технічного лідера.

**Обов'язки технічного лідера** (tech lead):
1. Архітектура й стек технологій проєкту.
2. Перехресне код рев'ю.
3. Технічний менторінг спеціалістів.
4. Введення стандартів коду.
5. Налаштування CI/CD (з DevOps).
6. Технічна документація.

Технічний лідер та Лідер команди — це ролі, а не рівень експертизи. В команді можуть бути всі спеціалісти високого рівня, але тільки один з них виконує роль технічного лідера.

Найм працівників виконує тех лідер разом з тім лідером. (Якщо тім лід і тех лід є однією особою, тоді bus factor збільшується, і можливість зірвати терміни релізу, deadline, також).

Лідер команди (тім лідер) повинен добре комунікувати з командою, залучати технічного лідера для обговорення технічних аспектів, налагоджувати розвиток команди.

Задача тім лідера й тех лідера оцінювати рівень розробників. Для цього розробляється спеціальний список питань. Крім того, потрібно оцінювати якість роботи розробника. Менеджер скоріше за все не буде мати технічної компетенції, щоб оцінити швидкість і якість виконання задачі. Менеджер звернеться за оцінкою до лідера команди або технічного лідера. Велика помилка коли тім лідер, або менеджер оцінює рівень працівника без відповідної на те компетенції (тех лідер в допомогу, або незалежний консалтинг).

Програміст повинен аналізувати вимоги, давати естімацію (оцінку задач), робити декомпозицію задачі.
Професійний програміст повинен розуміти, що за методологією Agile ми не вимагаємо детального опису задачі (прояснення вимог переходить на сторону комунікації).

Також є формула оцінки якості виконання задачі перед етапом QA:
1 - (Час на виправлення багів / час початкової естімації).

Зарплату членів команди краще встановлювати за об'єктивними критеріями відповідно до рівня досвіду, складності посади, ринкової зарплати.
Абсолютна система грейдів (junior-ставка, middle-ставка, senior-ставка) хороша тим, що робить розвиток програмістів незалежним й мотивує їх, якщо вона відповідає ринковим стандартам та є адекватною. При такій системі грейдів, програмістів не має цікавити заробітна плата один одного. Є грейд, є вимоги, є ставка цього грейда. Якщо немає прописаних рівнів зарплати й станеться так, що розробник з меншою зарплатою, дізнається про колегу того ж рівня, але з більшою зарплатою, тоді швидше всього він буде працювати менше за свого колегу (це очевидно).

**Якості лідера**, які описані в книзі Генка Рейнвотера "Як пасти котів":
1. Системне мислення, тобто розуміння зв'язків.
2. Широке мислення всім інтелектом, баланс між логікою й естетикою.
3. Вміння аналізувати.
4. Самокритика, вміння вчитися на помилках.
5. Вміння делегувати завдання.
6. Вміння слухати.
7. Нормативна й чітка мова, вміння висловлюватися.
8. Розуміння процесів та цілей бізнесу.

**Антипатерни лідера команди:**
1. Диктатор і генерал.
"Диктатор". Девіз цих діячів — "я правий, бо правий". Диктатор намагається нав'язувати своїм підлеглим конкретні методи вирішення поставлених ними завдань. Насправді всім нам не заважає ретельно стежити за тим, щоб не піти цим шляхом. Лідерство дійсно передбачає демонстрацію співробітникам можливих способів роботи, але в цьому слід бути обережним, оскільки позбавляти підлеглих почуття причетності до створення продукту неприпустимо. Диктатор створює навколо себе атмосферу жаху, яка дозволяє співробітникам повною мірою виявляти свої здібності. Справжній лідер повинен створювати умови, в яких розробники могли б пишатися результатами своїх праць, а щоб їхня гордість втілювалася в хороших результатах, їм слід надавати певну свободу вибору засобів досягнення мети.
"Генерали" виявляють значну схожість із диктаторами, виявляючи у своїй діяльності ще більшу жорсткість. Якщо ви уявили себе генералом, а своїх підлеглих вважаєте солдатами, майте на увазі: битви із противником вам доведеться вести одному. Солдати, швидше за все, будуть сидіти десь у кутку, спокійно попиваючи каву. Протистоїть такому підходу стиль співпраці — і, ви знаєте, він має широке вживання просто тому, що він правильний. Під час війни співпраця рідко має місце. У сфері розробки програмного забезпечення, навпаки, важко уявити ситуацію, у якій встановленню відносин з урахуванням співробітництва щось перешкодило б.

2. "Всезнайка". Такий діяч щиро вірить у те, що йому відомо все про його роботу, роботу компанії, про конкретні завдання програмістів. Він буде нехтувати порадами інших. Буде часто робити помилки. (Як казали багато філософів, чим більше знаєш, тим краще бачиш безодню невідомого).

3. Альфа "ботан".
Людина, яка постійно вказує на чужі помилки, як на невігластво. Думає, що всі мають знати те, що він, і ще більше. Такий працівник просто не розуміє психологію людей і що люди різні.

4. "Розумник". Людина, яка постійно любить повторювати очевидні банальні істини, щоб показати свої знання. "Розумник" дуже гордий і честолюбний. Він, чи вона, завжди публічно вкаже на слабкості інших, щоб підняти свою самооцінку за рахунок інших.

Деякі причини для звільнення члена команди:
1. Саботаж роботи, свідоме затягування процесів.
2. Критика і не повага до проджект менеджера, або лідерів команди.
3. Постійне скидання відповідальності та задач на інших.
4. Очевидна байдужість до продукту, відстороненість.

Відомий принцип тестування каже: *Не плануйте тестування, виходячи з припущення, що помилок не буде виявлено.*
Але це не означає, що ви маєте робити припущення про те, що програміст не старанно працює. Завжди вважайте, що розробник свідомо не винен у виникненні помилок, інакше будуть конфлікти, тобто застосовуйте презумпцію невинуватості. Це так звана Attribution bias.

Командний й технічний лідер повинен контролювати, щоб розробники середнього рівня:
1. Розвивались, вивчали необхідні технології.
2. Не робили критичних помилок.
3. Не вводили експериментальні, не надійні технології.
4. Виконували свою роботу.

Якщо спеціаліст справляється зі своїми обов'язками, йому потрібно хвалити за хорошу роботу, щоб він бачив, що ви ним задоволені. Не варто думати, що хвалять тільки за екстраординарні здобутки. Такий зворотний зв'язок допомагає молодшому спеціалісту розуміти, що він на правильному шляху й усуває його тривогу.

Якщо технічний лідер повинен дороблятиробоу програміста середнього рівня, тоді це дуже поганий знак, щось йде не так. Задача технічного лідера не робити все за всіх, а делегувати обов'язки, контролювати процес й підказувати, якщо необхідною.

Менеджерам проєкту важливо розуміти, що навіть якщо технічний лідер робить тільки якусь спеціалізовану частину проєкту, а все інше роблять інші програмісти, яких він контролює, все одно технічному лідеру потрібна ширша експертиза, обсяг знань, що повинно враховуватися в заробітній платі.

Комунікація дуже важлива в команді.
Структура організації та її комунікаційні канали впливають на структуру розроблюваного продукту. Наприклад, якщо команда розробників розділена на кілька підгруп, кожна з яких має обмежені комунікаційні засоби з іншими, то архітектура програмного забезпечення, ймовірно, відобразить ці розділення.
Організації, які створюють системи (що означає проєктування або розробку програмного забезпечення), обмежені у своїх проєктах тими комунікаційними структурами, які вони використовують.

**Закон Конвея** - "Будь-яка організація що проєктує системи, з великою ймовірністю створить дизайн зі структурою, яка буде копією структури комунікації в компанії".
Закон Конвея можна довести математично. Наприклад, у вас обмежений час і спілкуватися ви можете тільки азбукою Морзе. В таких умовах людина буде шукати найкоротше рішення відповідно до обмежень комунікації. Інший приклад, вам потрібно пояснити щось по телефону, але ціна розмови дуже висока. Ви буде підбирати найкоротшу форму пояснення, щоб досягнути цілі. Ці приклади

показують, що обмеження в комунікації встановлюють обмеження на рішення, відповідно й на архітектуру.

**PMBOK каже**: “
Однією з цілей ефективного лідерства є створення високоефективної проєктної команди. Існує кілька факторів, які сприяють створенню високоефективних проєктних команд:

Відкрита комунікація. Середовище, яке сприяє відкритій і безпечній комунікації, дозволяє проводити продуктивні наради, розв'язувати проблеми, генерувати ідеї тощо. Це також є основою для інших факторів, таких як спільне розуміння, довіра та співпраця.

Спільне розуміння. Мета проєкту та переваги, які він принесе, є загальними для всіх учасників.

Спільна відповідальність. Чим більше учасники проєктної команди відчувають відповідальність за результати, тим краще вони, ймовірно, будуть працювати.

Довіра. Проєктна команда, члени якої довіряють один одному, готова докладати додаткових зусиль для досягнення успіху. Люди менш схильні виконувати додаткову роботу, необхідну для успіху, якщо вони не довіряють своїм колегам по проєкту, керівнику проєкту або організації.

Співпраця. Проєктні команди, які співпрацюють і працюють разом, а не ізольовано або в конкуренції, як правило, генерують більш різноманітні ідеї та досягають кращих результатів.

Адаптивність. Проєктні команди, які здатні адаптувати свою роботу до середовища та ситуації, є більш ефективними.

Стійкість. Коли виникають проблеми або невдачі, високоефективні проєктні команди швидко відновлюються.

Наділення повноваженнями. Члени проєктної команди, які відчувають себе наділеними повноваженнями приймати рішення щодо своєї роботи, працюють краще, ніж ті, хто піддається мікроменеджменту.

Визнання. Проєктні команди, які отримують визнання за свою роботу та досягнуті результати, швидше за все, продовжуватимуть працювати добре. Навіть простий акт вираження вдячності підсилює позитивну поведінку команди”.

"Проєкт — тимчасове зусилля, здійснене для створення унікального продукту, послуги або результату. Тимчасовий характер проєктів передбачає початок і кінець роботи над проєктом або його фазою. Проєкти можуть бути самостійними або частиною програми чи портфоліо.

Портфоліо — проєкти, програми, дочірні портфоліо та операції, якими керують як групою для досягнення стратегічних цілей.

Продукт — артефакт, який виробляється, є кількісно визначеним і може бути як кінцевим продуктом, так і складовою частиною.

Керівник проєкту (Менеджер проєкту) — особа, призначена організацією, що виконує проєкт, для керівництва командою проєкту, яка відповідає за досягнення цілей проєкту. Керівники проєктів виконують різноманітні функції, такі як сприяння роботі команди проєкту для досягнення результатів і управління процесами для забезпечення запланованих результатів" (PMBOK Guide)

У своїй колекції нарисів про невдалі програмні проєкти Роберт Гласс (Robert Glass) склав список найбільш поширених "програмних катастроф", зокрема:
1. Неадекватний опис завдань проєкту.
2. Незадовільне планування та оцінка.
3. Застосування нової для цієї компанії технології.
4. Непридатна/відсутня методологія керівництва проєктом.
5. Нестача провідних спеціалістів групи.
6. Зрив домовленостей виробниками апаратного/програмного забезпечення.

**Bus factor** (фактор автобуса) проєкту — це кількість ключових учасників команди, які у випадку втрати своєї дієздатності, призведуть до неможливості рухати спільний проєкт далі. Bus factor є мірою зосередження інформації поміж окремими учасниками проєкту. Високий bus factor означає, що проєкт зможе продовжувати розвиватись навіть за несприятливих умов.
Вищий bus factor означає більшу стійкість проєкту до втрати ключових фахівців.
Збалансована розподіленість відповідальності та знань допомагає знизити цей ризик, забезпечуючи більшу стійкість команди до втрати будь-якого окремого члена.

Для зменшення "bus factor" у Scrum, рекомендується:

1. Робити code reviews: Забезпечте, щоб код переглядали інші члени команди, це допомагає із взаєморозумінням коду.

2. Документація: Створюйте докладну технічну документацію, щоб інші розробники могли легко розуміти та працювати з вашим кодом.

3. Навчання команди: Проводьте тренінги та діліться знаннями в команді, щоб інші члени могли взяти на себе роль у випадку необхідності.

4. Ротація завдань: Дозволяйте різним членам команди працювати з різними частинами проєкту, щоб було більше людей, які розуміють всю систему.

Ці підходи допоможуть розподілити знання і знизити ризик через "bus factor".

Робота вашого менеджера полягає в тому, щоб робити найкраще для компанії та команди, а не робити все можливе, щоб зробити вас щасливими.
Ваш керівник очікує, що ви принесете рішення, а не проблеми.

Артефакти в менеджменті — це конкретні об'єкти, документи або діаграми, що виникають у процесі управління проєктами чи бізнес-процесами. Це може включати плани, звіти, графіки Ганта та інші матеріали, які використовуються для організації та контролю проєктів. Артефакти грають важливу роль у забезпеченні чіткості та ефективності управління.

**MSCW метод** — це техніка пріоритизації, яка використовується у проєктному менеджменті для визначення важливості вимог в проєкті. Цей метод названий за абревіатурою з англійських слів Must, Should, Could та Won’t, які представляють різні рівні пріоритетів:

1. Must have (Має бути) - це обов'язкові елементи та умови, без яких проєкт не може вважатися завершеним або успішним. Вони є критичними для успіху.

2. Should have (Повинно бути) - це важливі елементи, які мають велику цінність для проєкту, але їх відсутність не є критичною.

3. Could have (Могло бути) - це бажані елементи, які могли б бути включені, якщо дозволяє час та ресурси. Вони не є критичними та можуть бути легко відкладені.

4. Won’t have (Не буде) - це елементи, які зараз не розглядаються або не включаються в поточний

обсяг проєкту.

Метод MSCW допомагає командам визначити, на чому слід зосередитись у першу чергу під час реалізації проєкту, а також дозволяє ефективно управляти очікуваннями стейкхолдерів щодо результатів проєкту.

**Маніфест гнучкої розробки** (Agile):

1. Люди та співпраця важливіші за процеси та інструменти.
2. Працюючий продукт важливіший за вичерпну документацію.
3. Співпраця із замовником важливіша за обговорення умов контракту.
4. Готовність до змін важливіша за дотримання плану.

Мікросервісна архітектура є підходом до розробки програмного забезпечення, в якому програма розбивається на невеликі, автономні модулі, відомі як мікросервіси. Кожен мікросервіс виконує конкретну функцію і може комунікувати з іншими мікросервісами через мережу. Цей підхід дозволяє полегшити розгортання, масштабування та розвиток програмного забезпечення.
Кожен мікросервіс може бути розроблений та підтримуватися окремою командою розробників.

Перехресне рев'ю коду — практика перегляду коду, коли рев'ю коду можуть робити учасники команди будь-якого рівня (а не тільки розробники однакового рівня).

"Перехід на особистості" — риторичний прийом, коли аргумент направлений не проти самої думки чи позиції, а проти особи, яка її висловлює. Замість того, щоб вести обговорення на рівні ідей чи доказів, людина, що використовує аргумент "Перехід на особистості", намагається підірвати позицію противника, звертаючись до його особистих характеристик, недоліків або вад. Це може бути використано для відволікання від фактів або справжньої суті обговорення.

Приклади:

1. "Я не вірю тобі щодо цієї проблеми, ти ж завжди робиш помилки",
2. "Цей вчений не вартий уваги, він не має навіть докторського ступеня",
3. "Я не підтримую твою позицію, ти ж ще молодий і не маєш досвіду".

**Атрибутивне упередження (Attribution Bias)** – це когнітивне упередження, яке виникає, коли люди схильні приписувати причини подій або поведінки інших людей на основі власних упереджень або обмеженого сприйняття, а не на основі об'єктивних фактів.
Схильність приписувати поведінку інших людей їх внутрішнім характеристикам (таким як особистісні риси або наміри), а не зовнішнім ситуаційним факторам. Наприклад, якщо хтось зробив помилку, ми можемо думати, що це через його некомпетентність, а не через складні обставини.

**Вирішення конфліктів** ефективно включає кілька кроків:

1. Визнання конфлікту: усвідомте, що є конфлікт і будьте готові його вирішити.

2. Зберігайте спокій та зосередженість: контролюйте емоції, щоб чітко і конструктивно спілкуватися.

3. Розумійте всі перспективи: слухайте всіх учасників, щоб зрозуміти їхні точки зору та занепокоєння.

4. Зосередьтеся на проблемі, а не на особі: адресуйте конкретну проблему, а не особистісні риси.

5. Знайдіть спільну мову: визначте спільні цілі чи інтереси, щоб побудувати основу для домовленості.

6. Генеруйте рішення: спільно працюйте над можливими рішеннями, заохочуючи креативність і відкритість.

7. Оцінюйте рішення: аналізуйте доцільність та вплив кожного рішення, щоб знайти найкращий варіант.

8. Дійдіть згоди щодо рішення: вирішіть, яке рішення можуть прийняти та на яке зможуть зобов'язатися всі сторони.

9. Впроваджуйте рішення: реалізуйте обране рішення на практиці та моніторьте його ефективність.

10. Здійснюйте перевірку: перевіряйте через деякий час, чи було конфлікт вирішено і чи працює рішення. За потреби вносьте корективи.

Книги для лідерів команди:

1. "Herding Cats: A Primer for Programmers Who Lead Programmers" by Hank Rainwater,

2. "The Mythical Man-Month" by Fred Brooks,

3. "The Manager's Path: A Guide for Tech Leaders" by Camille Fournier,

4. "Clean Agile: Back to Basics" by Robert Martin,

5. "Scrum: The Art of Doing Twice the Work in Half the Time" by Jeff Sutherland,

6. "Learning Agile: Understanding Scrum, XP, Lean, and Kanban" by Andrew Stellman,

7. "Extreme Programming Explained" by Kent Beck.

Agile full stack:

"Agile Software Development: Principles, Patterns, and Practices" by Robert C. Martin,

"Agile Estimating and Planning" by Mike Cohn,

"Agile Testing: A Practical Guide for Testers and Agile Teams" by Lisa Crispin,

"Agile Project Management For Dummies" by Mark C. Layton,

"Agile Project Management" by Jeffrey Ries,

"Clean Agile: Back to Basics" by Robert Cecil Martin,

Extreme programming:

"Extreme Programming Explained" by Kent Beck,

"Testing Extreme Programming" by Lisa Crispin,

Scrum:

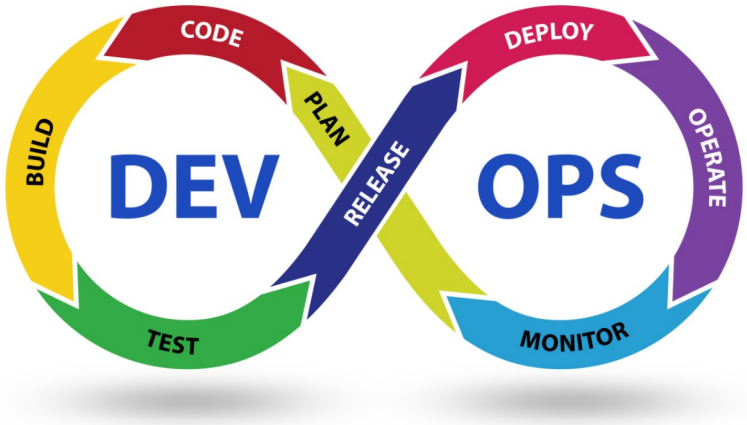
"Scrum guide" by Ken Schwaber and Jeff Sutherland,

"Essential Scrum: A Practical Guide to the Most Popular Agile Process" by Kenneth Rubin.

https://www.scrum.org/

# Шаблон команди та основних конвенцій

Хочете підвищити ймовірність успіху проєкту, найміть мінімум двох Сеньйорів в команду, також бажано Сеньйора QA, найміть Сеньйор Скрам майстра, і ще досвідченого DevOps спеціаліста. Чітко пропишіть, що якість коду має бути одним з приймальних критеріїв. Покривайте код автоматичними тестами, з боку розробників та QA. В команді має бути тех-лідер, який постійно слідкує за якістю коду. Чітко виконуйте всі Скрам церемонії. KPI визначається на основі цінності інкременту в спринті. Налаштуйте тестове середовище, та CI/CD.




Як лідер команди/технічний лідер переконайтеся, що ці процеси налаштовані ретельно:

-Scrum, XP (Story points estimation, team communication, retrospective, collective ownership, commitment).
Remember Conway's law, Brooks's law, 5 Scrum values.

-GitFlow. (Monorepo or polyrepo)

-Branch naming convention.

-Commit message convention.

-Code review guide (code style guide).

-CI/CD. Continuous Integration (CI) - is a practice, not just a tool.
Create a server for the backend, so frontend developers wouldn't have to set up the database locally after each change of backend.
(AWS CodePipeline, GitLab, GitHub).

-Jira flow (task board flow).

-Project tech documentation (Swagger, GraphiQL).

-Project (app) installation documentation.

-Onboarding documentation.

-App versioning (semver.org).

-Check “Bus factor”.

Programmers in team need:

• Requirements (Figma, PS, Confluence, Google Docs, SmartDocs).
• UI/UX Design approved by Programmer.
• Git server, CI/CD (GitLab, Bitbucket, Digital Ocean, Heroku, AWS).
• Place for communication (Slack, Skype, Google Meet, Zoom, Microsoft).
• Estimation (Pocker and T-shirt estimation).
• System for tracking task status (Jira).
• Methodology (Waterfall, Agile).
• Code standards, Code review.
• Testing, unit test (Jest), E2E tests (Cypress, Playwright).

У своїй колекції нарисів про невдалі програмні проєкти Роберт Гласс (Robert Glass) склав список найбільш поширених "програмних катастроф", зокрема:
1. Неадекватне специфікування завдань проєкту (51%).
2. Незадовільне планування та оцінка (48 %).
3. Застосування нової для цієї компанії технології (45%).
4. Непридатна/відсутня методологія керівництва проєктом (42%).
5. Нестача провідних спеціалістів групи (42%).
6. Зрив домовленостей виробниками апаратного/програмного забезпечення (42%)

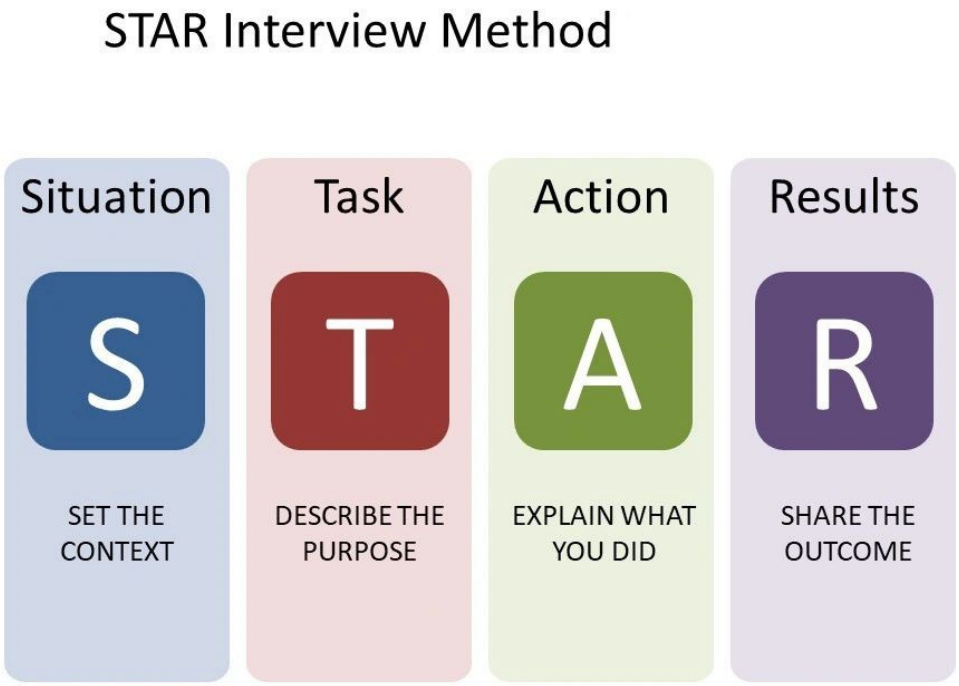
Джерела:

https://www.conventionalcommits.org/en/v1.0.0/
https://nvie.com/posts/a-successful-git-branching-model/
http://www.extremeprogramming.org/rules.html
https://scrumguides.org/
https://semver.org/
https://dev.to/varbsan/a-simplified-convention-for-naming-branches-and-commits-in-git-il4
https://google.github.io/styleguide/

Books that will help a team leader.

1. "Herding Cats: A Primer for Programmers Who Lead Programmers" by Hank Rainwater,
2. "The Mythical Man-Month" by Fred Brooks,
3. "The Manager's Path: A Guide for Tech Leaders" by Camille Fournier,
4. "Clean Agile: Back to Basics" by Robert Martin,
5. "Scrum: The Art of Doing Twice the Work in Half the Time" by Jeff Sutherland.

# Про методи співбесіди




1. Питання про досвід. Про складні задачі.

Плюси: Дають в загальному зрозуміти рівень кандидата.
Мінуси: Не завжди релевантні, тобто не дають змогу точно оцінити.

2. Постановка закритих питань (бінарних питань), які потребують короткої та однозначної відповіді.

Плюси: Легко виявити помилку.
Мінуси: Не показують цілісність знань та взаємозв'язки між ними.

3. Відкриті питання (open-ended question).

Плюси: Можуть виявити цілісність теоретичних знань.
Мінуси: Потребують більше часу та аналізу.

4. Ситуативне інтерв'ю: під час інтерв'ю кандидату пропонується вирішити змодельоване завдання, певну ситуацію (наприклад, вихід за межі спринту, або збір вимог).

Плюси: Показує практичні вміння кандидата.

5. Лайф кодинг.

Плюси: виявляє практичні навички кандидата, та його вміння.
Мінуси: Потребує багато часу і не точно відповідає умовам реальної роботи.

6. Метод STAR — це техніка для відповідей на співбесіді, що включає чотири елементи: Ситуація (опис контексту), Завдання (виклик, який виник), Дія (конкретні кроки, які ви зробили), та Результат

(ваші досягнення або вплив дій).

S – situation – ситуація,

T – task – завдання,

A – action – дія,

R – result – результат.

Метод STAR найдоречніше застосовувати при відповідях на запитання, які починаються зі слів: "Опишіть ситуацію, коли ви…".

Тривалість співбесіди бажано, щоб була не більше двох годин, краще година, півтори (тобто дві академічні години).

Іноді декілька сеньйорів не можуть знайти спільну мову та ефективно працювати в межах однієї команди. Ці питання допоможуть збільшити шанси знайти того, з ким вам буде комфортно працювати.

1. Запитайте про досвід, команду, в якій працював кандидат. Чи менторив він когось?

2. Дізнайтесь, як кандидат потрапив у професію та чому вона його цікавить.

3. Попросіть розробника написати програму "Hello World". Для чого це? Щоб побачити, який у нього редактор коду. Обговоріть редактор: чому саме він, які розширення встановлені. Редактор у команді має бути однаковим у всіх. Нагадаю, що VSCode та WebStorm підтримують dev containers.

4. Візьміть шматок свого хорошого коду та обговоріть його з розробником, проведіть разом код-рев'ю. Це одразу покаже стиль розробника, його підходи та широту поглядів (або навпаки – рамкове мислення і догматизм).

5. Запитайте, як розробник уявляє структуру команди. Як він бачить вирішення конфліктних ситуацій, коли між розробниками виникають розбіжності.

6. Запитайте, чи розробник розуміє п'ять цінностей Scrum.

7. Дізнайтесь, як розробник бачить співпрацю з QA-спеціалістом і що він очікує від нього.

8. Запитайте, які підходи в тій чи іншій сфері він вважає найкращими.

9. Дізнайтесь, чи готовий розробник працювати з вашим поточним стеком технологій та практиками.

10. Запитайте, як він розвивається технічно і які джерела використовує.

# Софт скіли — базові знання

Перший крок до спілкування — бажання почути думку. Якщо людина хоче Echo chamber, це не спілкування, максимум Confirmation bias.

Також важливо не робити Attribution bias, дивіться на реальні обставини.

Пасивна агресія та вислови з підтекстом (loaded questions) дуже шкідливі.
Пасивно-агресивна поведінка характеризується пасивною ворожістю та уникненням прямого спілкування. Замість прямих висловлювань, співрозмовник говорить (або пише) з негативним підтекстом.

В принципі, 5 Scrum values мають налаштувати здорову атмосферу в команді.

Успішне використання Скрам залежить від того, наскільки вміло люди втілюють наступні п’ять принципів:
Почуття обов'язку (Commitment),
Зосередженість (Focus),
Відкритість (Openness),
Повага (Respect) та сміливість (Courage).

Дивіться Scrum guide в перекладі Тетяни Островерх.

# Software Engineering Competency Model

**Software Engineering Competency Model (SWECOM, IEEE)** is organized by skill area (for example, software requirements), skills within skill areas (for example, software requirements elicitation), and activities within skills (for example, prototyping to elicit requirements). Activities are specified at five levels of competency:

• Technician

• Entry Level Practitioner

• Practitioner

• Technical Leader

• Senior Software Engineer

In general, a Technician follows instructions, an Entry Level Practitioner assists in performance of an activity or performs an activity with supervision; a Practitioner performs activities with little or no supervision; a Technical Leader leads individuals and teams in the performance of activities; and a Senior Software Engineer modifies existing methods and tools and creates new ones. Some organizations may choose to merge the Technician and Entry Level Practitioner levels. A Senior Software Engineer may serve as a "chief engineer" for a software organization and some Senior Software Engineers may be recognized as industry experts who contribute to shaping and advancing the profession of software engineering.

In addition to the activities specified at the various competency levels, an additional competency of all software engineers is to instruct and mentor others, as appropriate, in the methods, tools, and techniques used to accomplish those activities. For example, a Technician or Entry Level Practitioner might instruct or mentor others on the use of configuration management tools as needed to perform their activities, or a Team Leader might instruct or mentor a Practitioner on how to lead inspections and reviews.

The following notations are also used in SWECOM:

• Follows (F),

• Assists (A),

• Participates (P),

• Leads (L), and

• Creates (C).

For the requirements prototyping activity cited above, a Technician would be competent to use software tools while following instructions (F) to create prototypes. An Entry Level Practitioner would be competent to assist in creating prototypes and to develop prototypes under supervision (A); a Practitioner would create prototypes and interact with customers and users in evaluating the prototypes (P); a Technical Leader would supervise and lead prototyping activities (L); and a Senior Software Engineer would create new approaches to prototyping (C).

There may be situations where an Entry Level Practitioner might be competent, for example, to lead a prototyping activity, or a Technical Leader might be competent to create a new approach to prototyping, so notations are used in SWECOM to distinguish specific competencies from the named competency levels when it is appropriate.

A Practitioner, for example, might be competent to either participate in an activity (P) or lead the activity (L), depending on the scope and complexity of the work to be accomplished. In this case, the activity is labeled (P/L) at the Practitioner level. Similarly, an Entry Level Practitioner might be competent to assist or fully participate in an activity, which would be labeled (A/P).

SWECOM does not prescribe the knowledge level or years of experience associated with these competency levels; however, the following general guidelines are typical:

An individual who is competent at the Technician level to perform the activities in one or more skills or skill areas might have some advanced education (for example, a two-year US associate's degree or equivalent), one or more industrial certifications, and any number of years of experience.

An individual who is competent as an Entry Level Practitioner to perform the activities in one or more skills or skill areas would probably have requisite knowledge equivalent2 to that provided by an ABET-accredited software engineering degree program or equivalent and zero to four or five years of relevant experience.3

An individual who is competent at the Practitioner level to perform the activities in one or more skills or skill areas would probably have knowledge equivalent to or greater than that of an Entry Level Practitioner, might have a master's degree in software engineering or a related discipline, and would probably have more than five years of experience in the relevant skill areas.

An individual who is competent as a Technical Leader for one or more SWECOM activities, skills, or skill areas would likely have relevant knowledge and experience equal to or greater than that of a Practitioner plus the behavioral attributes and skills needed to be an effective technical leader.

A Senior Software Engineer is an individual who is competent to develop policies, procedures, and guidelines for the technical processes and work products within an organizational unit that is engaged in software engineering.

# Методології розробки програмного забезпечення

## Що таке Scrum?

Скрам був розроблений насамперед як фреймворк для розробки програмного забезпечення. Він враховував попередні напрацювання інженерів в таких методологіях як FDD та XP. Можна сказати, що шлях до Скраму та Extreme Programming (XP) почався в 1960-х роках, коли виник “Software crisis” та стихійна (хаотична) розробка з мікроменеджментом.

Scrum (Скрам) та Extreme Programming (XP) ґрунтуються на Agile маніфесті.

Перша редакція Agile маніфесту була написана з 11 по 13 лютого 2001, на гірськолижному курорті в горах Юти.
Серед авторів маніфесту були: Кент Бек, Роберт Мартін, Мартін Фаулер, Ендрю Хант, Дейв Томас, Кен Швабер, Джефф Сазерленд.

**Маніфест для Agile** розробки програмного забезпечення:

1. Люди та співпраця важливіші за процеси та інструменти.
2. Працюючий продукт важливіший за вичерпну документацію.
3. Співпраця із замовником важливіша за обговорення умов контракту.
4. Готовність до змін важливіша за дотримання плану.

Гнучкі методики розробки (agile software development, Agile розробки) — узагальнюючий термін для цілого ряду підходів і практик, що ґрунтуються на цінностях Маніфесту гнучкої розробки програмного забезпечення та 12 принципах, що лежать у його основі.

Scrum — це легкий фреймворк, який допомагає людям, командам та організаціям створювати цінність шляхом адаптивних рішень для складних проблем.
Кен Швабер та Джефф Сазерленд вперше спільно представили Scrum на конференції OOPSLA у 1995 році.

П'ять цінностей Скраму: **Сміливість** — приймати виклики, **Зосередженість** — концентруватися на важливому, **Відданість (**commitment**)** — відповідальність за досягнення цілей, **Відкритість** — прозорість у спілкуванні, **Повага** — цінування внеску кожного члена команди.

Фреймворк — це каркас, або система принципів, яка встановлює певні рамки для роботи.

**Ролі в Scrum:**

- Власник продукту (Product Owner): Представляє клієнта та визначає, що потрібно будувати.
- Скрам-майстер (Scrum master): Налагоджує процес Scrum і допомагає команді подолати перешкоди.
- Команда розробників: Крос-функціональна команда, відповідальна за доставку продукту.

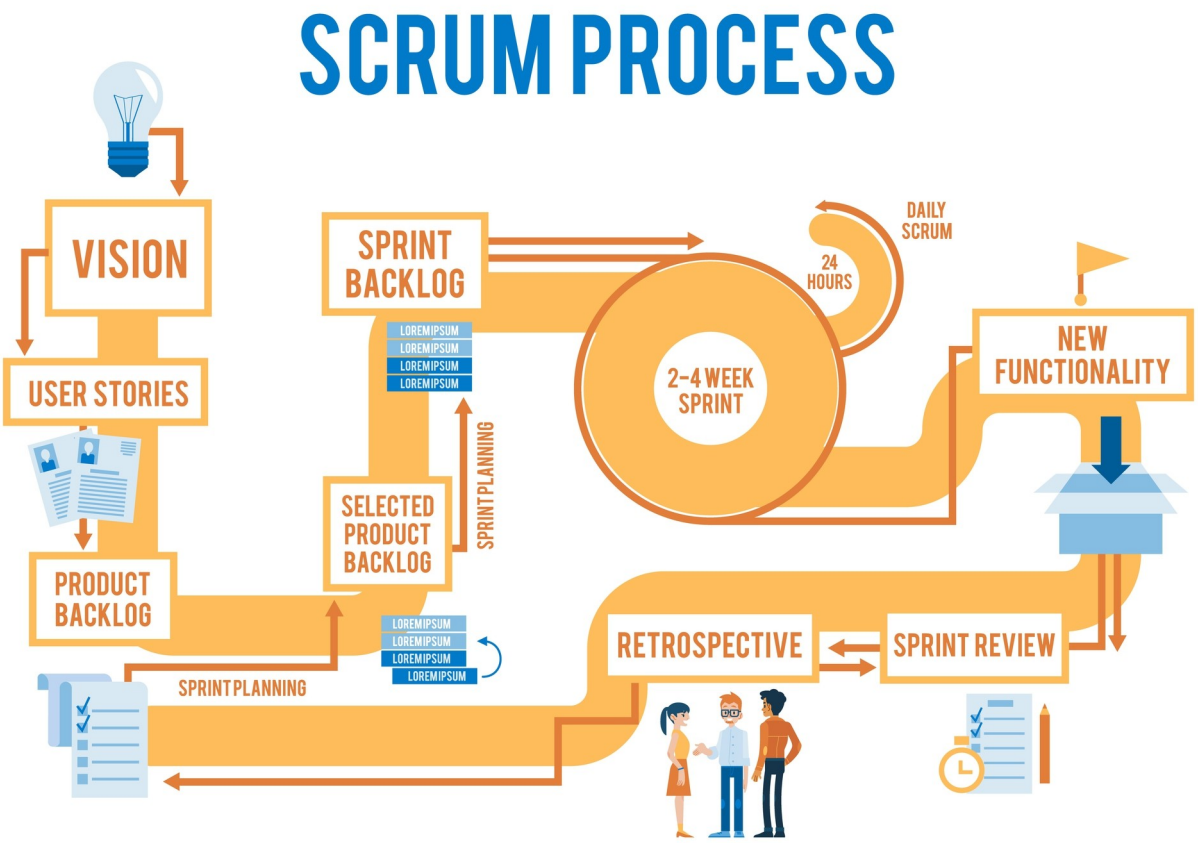
Найефективнішим і дієвим методом передачі інформації команді розробників і всередині неї є бесіда віч-на-віч.

Команди Scrum є крос-функціональними, що означає, що учасники мають всі необхідні навички для створення цінності кожного спринту.
Колаборативний характер крос-функціональної команди Scrum сприяє відчуттю спільної відповідальності.
Замість того, щоб покладатися на одну особу для вирішення певного завдання чи області, команда спільно бере на себе власність і співпрацює, щоб вирішити будь-які перешкоди чи проблеми, що

виникають.
Хоча крос-функціональні команди призначені мати різноманітний набір навичок, це не означає, що кожен учасник повинен бути експертом у всіх областях.
Крос-функціональні команди призначені для концентрації на одному проєкті або продукті одночасно.
Ефективна координація та комунікація все ще є ключовими в крос-функціональних командах.
Команда Scrum складається з одного Scrum Master, одного Product Owner та розробників. У команді Scrum немає підкоманд чи ієрархій. Це згуртований підрозділ професіоналів, які зосереджені на одній меті за раз, цілі продукту.
Хоча формальна ієрархія відсутня, технічний лідер може виконувати роль старшого фахівця, який підтримує команду своїми знаннями та досвідом. Це допомагає забезпечити високий технічний рівень рішень.
Технічний лідер, який часто називається "Tech Lead", може існувати в Scrum команді для керування технічним боком проєкту. Він відповідає за архітектуру та технічне бачення, допомагає команді у розв'язані складних технічних проблем та забезпечує якість коду.

Scrum вимагає від Скрам-майстра створення середовища, де:

1. Власник продукту розміщує роботу для складної проблеми у Backlog продукту.
2. Команда Scrum перетворює відібрану роботу на приріст вартості (Інкремент цінності) під час спринту.
3. Команда Scrum та її зацікавлені сторони (stakeholders) оглядають результати та вносять корективи на наступний спринт.
4. Процес повторюється.




Під час Scrum спринту ідеї перетворюються на цінність.
Спринти — це фіксовані за часом події тривалістю один місяць або менше (2 тижні).
Новий спринт починається негайно після завершення попереднього.

Весь необхідний обсяг роботи для досягнення мети продукту, включаючи планування спринту, щоденні Scrum-зустрічі, огляд спринту та ретроспективу, відбуваються протягом спринтів.

Під час спринту:
- Не вносяться зміни, які б могли загрожувати меті спринту;
- Якість не знижується;
- Беклог (backlog) продукту уточнюється за потреби; і,
-Об'єм (scope) може бути уточнений та переглянутий з Власником продукту, коли стане відомо більше.

*Спринт можна скасувати, якщо ціль спринту застаріє. Лише Власник продукту має право скасувати спринт.*

Зазвичай проєкт у методології Scrum дотримується цих правил:
Кожен спринт починається з планування, проведеного Скрам-майстром, Власником продукту та рештою команди, і складається з наради (meeting), розділеної на дві частини, кожна з яких обмежена в часі чотирма годинами. Домашнє завдання Власника продукту перед спринт-плануванням — розробити пріоритетний беклог продукту, що складається з набору елементів (items), які користувачі та зацікавлені сторони погодили. У першій частині наради (зустрічі) Власник продукту співпрацює з командою, щоб вибрати елементи, які будуть доставлені до кінця спринту на основі їхньої цінності та оцінки роботи, яку вони вимагають. Команда згодна продемонструвати робоче програмне забезпечення, що включає ці елементи, до кінця спринту. Ця перша частина обмежена в часі (для спринту тривалістю 30 днів вона обмежена в часі чотирма годинами; для коротших спринтів вона пропорційно коротша), так що в кінці команда бере все, що вони вже зробили, і використовує це як беклог спринту. У другій частині наради члени команди (з допомогою Власника продукту) визначають індивідуальні завдання, які вони використовуватимуть для реальної реалізації цих елементів. Знову ж таки, ця частина обмежена в часі в залежності від тривалості спринту. На кінець планування спринту вибрані елементи стають беклогом спринту.

Команда проводить щоденну Scrum нараду (Daily Scrum meeting) щодня. Всі члени команди (включаючи Scrum-майстра та Власника Продукту) повинні брати участь, а зацікавлені сторони (stakeholders) також можуть брати участь (але повинні залишатися тихими спостерігачами). Нарада обмежена в часі 15 хвилин, тому всі члени команди повинні приходити вчасно. Кожен член команди відповідає на три питання: Що я зробив з моменту останньої щоденної наради? Що я збираюся зробити між цією та наступною щоденною нарадою? Які перешкоди та завади стоять на моєму шляху? Кожен член команди має бути лаконічним; *якщо відповідь потребує обговорення, відповідні члени команди планують наступну нараду відразу після наради.*
Дейлі мітинг (стендап) повинен бути не більше 15 хвилин. В ньому беруть участь "Розробники" - всі, хто робить щось для інкременту в спринті. Інші гості можуть бути хіба як спостерігачі, але на них час не виділяється.
Кожен учасник Дейлі мітингу каже, як його успіхи, чи все йде по плану. І коротко каже, які в нього виникли проблеми, щоб потім йому хтось допоміг, після мітингу. Дейлі мітинг (Дейлі Скрам) принципово не є мітингом для звіту, звітування, це мітинг для синхронізації.

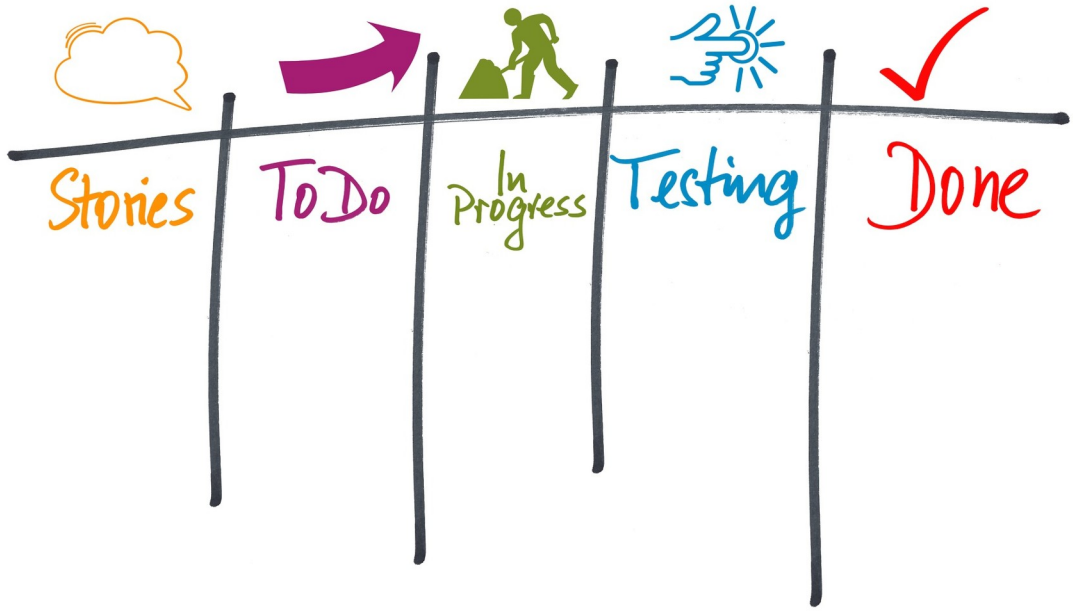
Кожний спринт обмежений в часі певною тривалістю, визначеною під час планування спринту: багато команд використовують 30 календарних днів, але ця тривалість може відрізнятися — деякі команди обирають двотижневі спринти. Протягом спринту команда перетворює елементи беклогу спринту в робоче програмне забезпечення. Вони можуть отримувати допомогу від людей, які не є членами команди, але такі люди не можуть диктувати команді, як виконувати їхню роботу, і повинні довіряти команді доставку. Якщо хтось з команди виявляє посередині спринту, що вони зобов'язалися занадто багато або що вони можуть ще додати елементи, вони повинні переконатися, що Власник продукту дізнається, як тільки вони розуміють, що спринт у небезпеці. Власник продукту (Product owner) — член команди, який може співпрацювати з користувачами та зацікавленими сторонами, щоб переглянути їх очікування, і використовувати цю інформацію для коригування беклогу спринту, щоб відповідати фактичній потужності команди. І якщо вони виявляють, що у них закінчиться робота до кінця спринту, вони можуть додати більше елементів до беклогу спринту. Команда повинна тримати беклогу спринту актуальним та видимим для всіх.  Product owner – це роль.

В дуже аномальних випадках та в екстремальних обставинах Власник продукту може припинити спринт раніше і почати нове планування спринту, якщо команда виявляє, що вони не можуть доставити робоче програмне забезпечення (наприклад, виникає серйозна технічна, організаційна або кадрова проблема). Але всі повинні знати, що припинення спринту є рідкісним явищем і має дуже негативні наслідки з точки зору їхньої здатності виробляти та доставляти програмне забезпечення.

Після завершення спринту команда проводить зустріч огляду спринту (sprint review meeting), де вони демонструють робоче програмне забезпечення користувачам та зацікавленим сторонам. Демонстрація може включати лише ті елементи, які фактично завершені та протестовані (що в цьому випадку означає, що команда завершила всю роботу над ним і протестувала її, і що це було прийнято Власником продукту як завершене). Команда може представляти лише функціональне, робоче програмне забезпечення, а не проміжні елементи, такі як схеми архітектури, схеми баз даних, функціональні специфікації й т.д. Зацікавлені сторони можуть ставити питання, на які команда може відповісти. На кінець демонстрації зацікавлені сторони запитуються про свої думки та відгуки, і є можливість поділитися своїми думками, почуттями, ідеями та думками. Якщо потрібні зміни, це враховується при плануванні наступного спринту. Власник продукту може додати зміни до беклогу продукту, і якщо вони потрібні негайно, вони потраплять до беклогу наступного спринту.

Після спринту команда проводить зустріч ретроспективи спринту (sprint retrospective meeting), щоб знайти конкретні шляхи поліпшення своєї роботи. Участь беруть команда та Scrum-майстер (і, за бажанням, Власник продукту). Кожна людина відповідає на два питання: Що пішло добре під час спринту? Що може покращитися у майбутньому? Scrum-майстер фіксує будь-які покращення, і конкретні елементи (такі як налаштування нового сервера збирання, впровадження нової практики програмування чи зміна офісного розташування) додаються до беклогу продукту.

У методології Scrum є поширений інструмент, який називається **"Scrum board" або "Kanban board"**. Це зазвичай цифрова дошка (або фізична дошка), розділена на колонки, що представляють різні етапи роботи, такі як "To Do", "In Progress" і "Done". Дошка допомагає візуалізувати прогрес завдань або користувацьких історій протягом спринту або проєкту, що полегшує команді відстеження їхньої роботи та виявлення будь-яких заторів або проблем.



Ємність (Capacity) — це максимальна кількість роботи, яку команда може взяти на себе в наступному спринті. Команда може швидко встановити ємність, виходячи з середньої швидкості, а потім коригувати її враховуючи доступність кожного під час наступного спринту.

Навантаження (Load) — це кількість роботи, обрана командою для поточного спринту. Це кількість роботи, яку команда планує завершити протягом спринту. Воно повинно бути менше або рівним ємності.

Швидкість (Velocity) — це кількість роботи, завершена в попередніх спринтах. Це міра минулої продуктивності команди. Зазвичай дивляться на останні три-п'ять спринтів і беруть середнє значення їхньої швидкості.

Ось формули:
швидкість >= ємність >= навантаження,
буфер = ємність - навантаження
Різниця між ємністю та навантаженням — це ваш планувальний буфер.

- Беклог продукту: Пріоритетний список функцій та покращень.
- Беклог спринту: Функції, обрані з Беклогу продукту для спринту.
- Інкремент: Сума всіх завершених завдань в кінці спринту.
- Спринт: Часовий блок (зазвичай 2-4 тижні) під час якого створюється потенційно готовий до відвантаження інкремент продукту.
- Планування спринту: Зустріч для планування роботи на майбутній спринт.
- Щоденний Scrum (Daily Scrum): Коротка щоденна зустріч для синхронізації та планування роботи на день.
- Огляд спринту (sprint review): Демонстрація завершеної роботи в кінці спринту.
- Ретроспектива спринту: Аналіз минулого спринту для покращення процесів.
Scrum підкреслює ітеративний та інкрементальний розвиток, сприяючи співпраці, гнучкості та постійному вдосконаленню.

Щоденний Scrum — це подія тривалістю 15 хвилин для розробників команди Scrum.

Огляд спринту — передостання подія спринту та обмежується до чотирьох годин для місячного спринту. Для коротших спринтів подія, зазвичай, коротша.

Мета ретроспективи спринту — спланувати способи підвищення якості та ефективності.

Беклог продукту — це впорядкований список того, що потрібно для поліпшення продукту. Це єдине джерело роботи, яке виконується командою Scrum.

Скрам в основному зосереджений на організаційних процесах розробки програмного забезпечення, але Екстремальне програмування (Extreme Programming, XP) навпаки, акцентується на процесах кодування та тестування. XP було розроблено Кентом Беком. Воно підкреслює керовану тестами розробку (Test-Driven Development, TDD), парне програмування, спільну власність, постійну інтеграцію/постійне розгортання (Continuous Integration/Continuous Deployment, CI/CD) та невеликі ітерації.

**Story Point** (Сторі пойнт) та міфічна людино-година.

Коли ми вводимо поняття "людино-година", ми робимо припущення: що всі люди (припустимо, інженери ПЗ) мають рівні вміння, а отже можна чітко визначити час на виконання певної задачі. Це припущення вже невірне, коли в команді є старші та молодші спеціалісти, які роблять одне й те саме за різний час. Крім того, поняття "людино-година" говорить про те, що збільшення кількості людей буде пропорційно відображатися на зменшенні терміну створення продукту, але це міф. Цю помилку розбирав Фред Брукс у своїй книзі "Міфічний людино-місяць". Фредерік Брукс пише, що потрібно врахувати витрати на комунікацію між інженерами, які в деяких випадках можуть бути настільки великі, що виконується закон Брукса: "Додавання робочої сили до запізнілого програмного проєкту

затягує його ще більше". Додавання більшої кількості людей до завдання, яке дуже поділене, наприклад, прибирання номерів у готелі, зменшує загальну тривалість завдання (аж до моменту, коли додаткові працівники заважають один одному). Однак інші завдання, включаючи багато спеціальностей у проєктах програмного забезпечення, менш подільні. Щоб вирішити ці недоліки поняття "людино-година", були створені story points в Scrum.

Story point (Очки складності завдання) - це одиниця виміру, яка використовується в методології розробки програмного забезпечення для оцінки складності задачі. Вона використовується для приблизної оцінки часу і зусиль, не прив'язуючись до конкретної часової рамки.

Коли команда розробників оцінює задачу, вони приймають у розгляд три основні фактори: складність, обсяг роботи та ризики. Вони потім призначають задачі певну кількість story points, яка відображає загальну складність. Чим більше story points, тим складніше завдання. Це дозволяє команді спрогнозувати, скільки робочих годин знадобиться на виконання задачі і планувати роботу відповідно.

Між story point та часом не має прямопропорційної залежності.

Коли ми вводимо поняття "**людино-година**", ми робимо припущення: що всі люди (припустимо, інженери ПЗ) мають однакові вміння, а отже можна чітко визначити час на виконання певної задачі. Це припущення вже невірне, коли в команді є старші та молодші спеціалісти, які роблять одне й те саме за різний час. Крім того, поняття "людино-година" говорить про те, що збільшення кількості людей буде пропорційно відображатися на зменшенні терміну створення продукту, але це міф. Цю помилку розбирав Фред Брукс у своїй книзі "Міфічний людино-місяць". Фредерік Брукс пише, що потрібно врахувати витрати на комунікацію між інженерами, які в деяких випадках можуть бути настільки великі, що виконується закон Брукса: "Додавання робочої сили до запізнілого програмного проєкту затягує його ще більше". Додавання більшої кількості людей до завдання, яке дуже поділене, наприклад, прибирання номерів у готелі, зменшує загальну тривалість завдання (аж до моменту, коли додаткові працівники заважають один одному). Однак інші завдання, включаючи багато спеціальностей у проєктах програмного забезпечення, менш подільні. Щоб вирішити ці недоліки поняття "людино-година", були створені story points в Scrum.

**Story point** (Очки складності завдання) - це одиниця виміру, яка використовується в методології розробки програмного забезпечення для оцінки складності задачі. Вона використовується для приблизної оцінки зусиль, не прив'язуючись до конкретної часової рамки.

Коли команда розробників оцінює задачу, вони приймають у розгляд три основні фактори: складність, обсяг роботи та ризики. Вони потім призначають задачі певну кількість story points, яка відображає загальну складність. Чим більше story points, тим складніше завдання. Це дозволяє команді спрогнозувати, скільки робочих годин знадобиться на виконання задачі та планувати роботу відповідно.

Між story point та часом не має прямопропорційної залежності.

З допомогою Story points визначається velocity of Scrum team.

**Аксіоми Story points** (Очки задачі):

1. Story points відносні (поставте якесь опорне завдання).
2. Story points представляють відносні зусилля, складність і ризики для завдання. (Складність стосується структури, а не часу).
3. Story points є лінійними: 2 SP = 1 SP * 2. (Одна 3-points story повинна вимагати приблизно стільки ж роботи, скільки інша 3-points story).
4. Не існує найбільшого числа сторіпойнтів (story points). (На практиці кладуть ліміт, наприклад, якщо одна задача більше ніж 13 сторіпойнтів, розділяємо її на дві).

Деталі про Story points в книгах:
"Agile Estimating and Planning" by Mike Cohn,
"Learning Agile" by Andrew Stellman.

**Nexus** - фреймворк для мульти-командної розробки в рамках Scrum.
Коли продукт дуже великий його ділять на модулі, мікросервіси, мікрофронтенд. Кожен модуль та мікросервіс розробляє окрема команда. Але потрібно це все інтегрувати та керувати цим процесом.

Методологія Nexus є однією з підходів до масштабування розробки Agile, спрямованою на спільну роботу декількох Scrum-команд над великими проєктами.

Основні правила методології Nexus включають:

-Роль Nexus Integration Team (NIT): Спеціалізована команда, яка відповідає за координацію роботи декількох Scrum-команд у межах Nexus.

-Nexus Sprint Planning: На цьому етапі *представники* всіх Scrum-команд розглядають інкременти, що плануються, і спільно визначають, які завдання потрібно виконати для досягнення цілей.

-Nexus Daily Scrum: Щоденні зустрічі для спілкування між членами команд та вирішення проблем, що можуть виникнути при інтеграції робочих інкрементів.

-Nexus Sprint Review: Під час цього заходу *представники* всіх команд демонструють свої робочі інкременти та обговорюють, які кроки слід вжити для подальшого вдосконалення.

-Nexus Sprint Retrospective: Регулярні огляди роботи всієї Nexus-структури з метою виявлення та виправлення проблем та покращення робочих процесів.
-Nexus Goal: Кожен спринт має свою мету, яка визначається спільно всіма командами та допомагає забезпечити спільний фокус на досягнення цілей.

-Nexus Daily Scrum of Scrums (DoS): Це щоденна зустріч *представників* кожної Scrum-команди з метою обговорення інтеграційних питань та вирішення конфліктів.

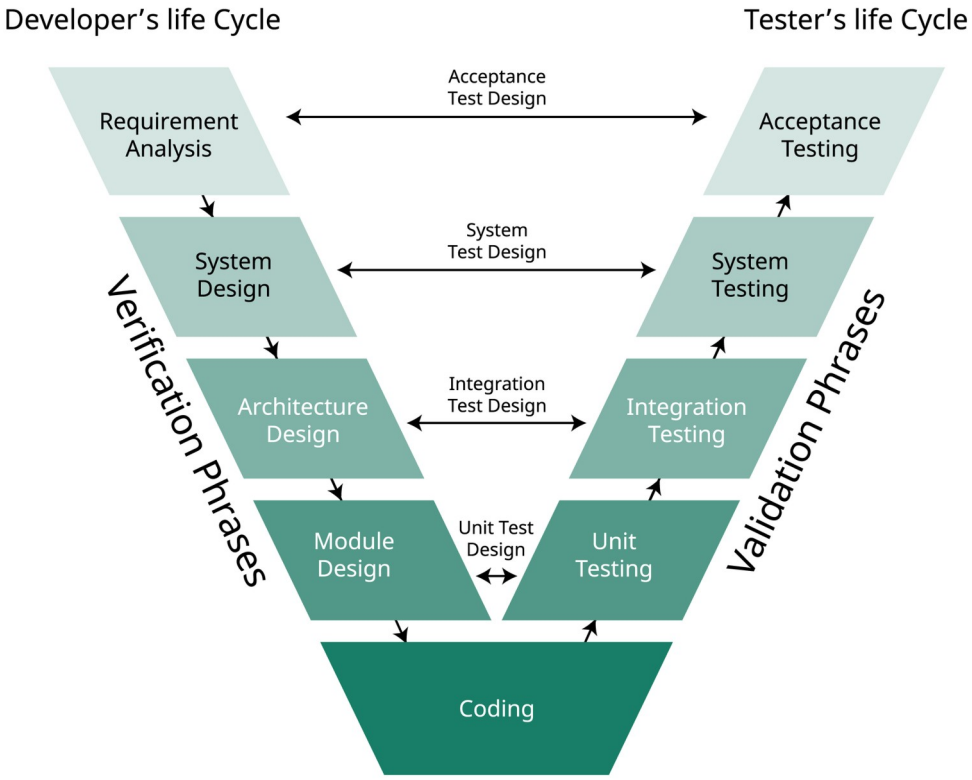
-Nexus Sprint Backlog: Спільний перелік завдань для всіх Scrum-команд, який допомагає уникнути дублювання роботи та забезпечити спільне розуміння пріоритетів.

Ці правила допомагають забезпечити ефективну спільну роботу декількох команд у межах Nexus, спрощуючи співпрацю та забезпечуючи координацію.

**Методологія Водоспад та V-модель**

Методологія Водоспад (Waterfall) — це традиційний підхід до управління проєктами, який використовується в розробці програмного забезпечення та інших галузях. Вона характеризується лінійним та послідовним процесом, де прогрес поступово рухається вниз (як водоспад) через попередньо визначені фази. Ось розбір типових етапів Водоспаду: Збір вимог, Проєктування системи, Реалізація, Тестування, Розгортання, Обслуговування. Одна з ключових характеристик методології Водоспад — кожна фаза повинна бути завершена перед переходом до наступної. Це означає, що мало місця для гнучкості або змін після завершення фази, що може бути обмеженням в проєктах, де вимоги ймовірно зміняться чи розвиватимуться з часом. Крім того, зворотний зв'язок від користувачів або зацікавлених сторін зазвичай приходить пізно в процесі, що може призвести до дорогоцінного перероблення, якщо виникають непорозуміння або зміни в вимогах. Незважаючи на ці недоліки, методологія Водоспад може бути ефективною для проєктів з чітко визначеними та стабільними вимогами, де важливі передбачуваність та контроль.
В моделі Waterfall тестування зазвичай відбувається після завершення всіх етапів розробки.
V-модель розширює модель Waterfall, додаючи тестування на кожному етапі розробки. Кожен етап розробки має відповідний етап тестування, утворюючи форму літери "V".

V-модель є концептуальною моделлю, що використовується в рамках методології Waterfall для візуалізації зв'язку між різними етапами проєкту та відповідними тестами. На початку V-діаграми розташовані етапи, такі як збір вимог, аналіз, проєктування системи та програмування, де визначаються вимоги та планується архітектура системи. Після цього настає фаза реалізації, де виконується написання коду. Після завершення фази реалізації, робота переходить до фази тестування, де виконуються тести для перевірки відповідності програмного забезпечення вимогам та специфікаціям. Всі ці етапи, починаючи з аналізу та завершуючи тестуванням, утворюють "прямий шлях" V-діаграми. У V-моделі тестування починається відразу після виконання кожного етапу розробки.




V-модель не відповідає Agile розробці.

**Екстремальне програмування**

Екстремальне програмування (Extreme Programming, XP) було розроблене в 1990-х роках Кентом Беком разом з іншими програмістами. XP є одним із методів гнучкої розробки програмного забезпечення, який акцентує на швидких ітераціях розробки, невеликих командних розмірах, тестуванні перед написанням коду та підтримці стандартів якості. (Book "Extreme Programming Explained" by Kent Beck, 1999)

Принципи Екстремального програмування (Extreme Programming, XP):

Планування:
- Користувацькі історії (user stories) написані.
- Планування випуску (Release planning) створює розклад випуску.
- Робіть часті та невеликі випуски.
- Проєкт розділений на ітерації.
- Планування ітерацій (Iteration planning) починається з кожної ітерації.

Управління:
- Зустріч “стендап” починається кожен день.
- Вимірюється швидкість проєкту (Project Velocity).
- Наднормові вважаються поганою практикою.

Проєктування:
- Простота (принцип KISS).
- Виберіть системну метафору, для загального опису системи.
- Створіть пробні рішення (spikes) для зменшення ризику.
- Жодна функціональність не додається заздалегідь (принцип YAGNI).
- Робіть рефакторинг, коли це можливо, де завгодно (але перед цим пишіть тести).

Кодування:
- Клієнт завжди доступний.
- Код повинен бути написаний відповідно до погоджених стандартів.
- Спочатку напишіть модульний тест (TDD).
- Усі виробничі коди програмуються в парі (pair programming).
- Тільки одна пара інтегрує код одночасно.
- Інтегруйте часто.
- Налаштуйте спеціальний інтеграційний комп'ютер (сервер). Автоматизація є основним принципом досягнення успіху DevOps, а CI/CD є критично важливим компонентом.
- Використовуйте колективну власність (collective ownership). За код відповідальна вся команда.

Тестування:
- Весь код повинен мати модульні тести.
- Весь код повинен пройти всі модульні тести, перш ніж його можна буде випустити.
- Якщо знайдено помилку, створюються тести.
- Приймальні тести (Acceptance tests) часто запускаються, і результати публікуються.

Простота: Ми будемо робити те, що потрібно та вимагається, але не більше. Це максимізує створену вартість від вкладених до цього дня інвестицій.

Комунікація: Кожен є частиною команди, і ми щоденно спілкуємося. Ми працюватимемо разом над усім, починаючи від вимог до коду.

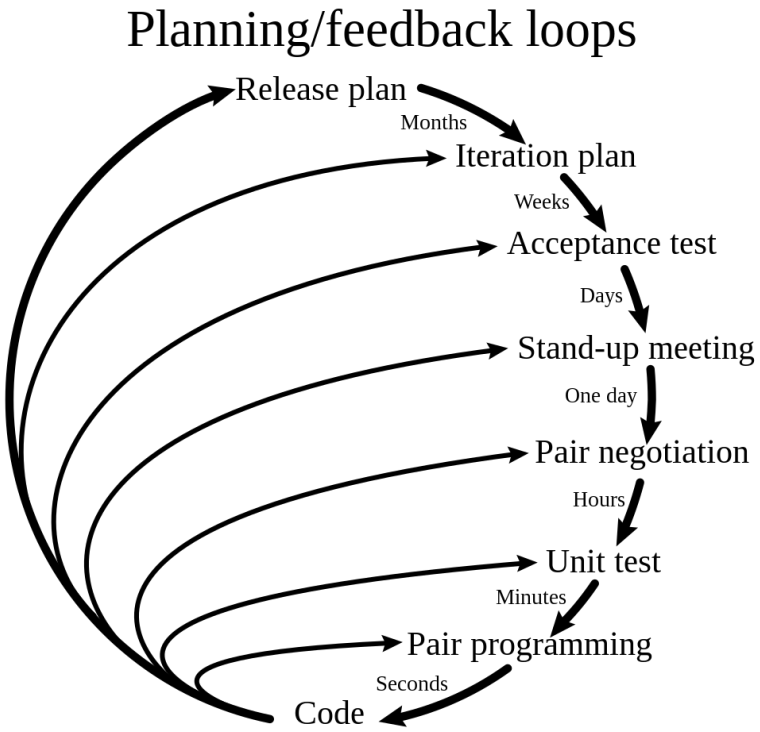
Зворотний зв'язок: Ми серйозно ставимось до кожної ітераційної зобов'язаності, поставляючи робоче

програмне забезпечення. Ми демонструємо наше програмне забезпечення часто та рано, потім уважно слухаємо та вносимо будь-які зміни, які необхідні.

Повага: Кожен отримує та відчуває повагу, яку він заслуговує як цінний член команди. Кожен вносить свою вартість, навіть якщо це просто ентузіазм. Розробники поважають експертизу клієнтів, і навпаки. Керівництво поважає наше право приймати відповідальність та отримувати повноваження над своєю роботою.

Мужність: Ми говоритимемо правду про прогрес та оцінки. Ми не документуємо виправдання для невдачі, оскільки ми плануємо успіх.

Метафора системи — це простий і послідовний спосіб опису вашої програмної системи за допомогою знайомої області або аналогії. Це допомагає вам і вашій команді спілкуватися та розуміти дизайн, архітектуру та функціональність вашої системи. Наприклад, якщо ви створюєте веб-бібліотечну систему, ви можете використати метафору фізичної бібліотеки з книгами, полицями, каталогами, позиками та поверненнями. Ця метафора може допомогти вам узгоджено називати ваші класи, методи, змінні та інтерфейси.




Спайк (spike) - це інновація в екстремальному програмуванні (XP) в гнучкій (agile) розробці програмного забезпечення. Це невелика історія (user story) або робота, яка призначена для збору інформації, а не для створення інкременту в продукті.

Слово "спайк" походить від скелелазних занять. Під час сходження ми можемо зупинитися, щоб вбити спайк (цвях) в скелі, що не є фактичним сходженням, але цим ми забезпечуємо, що майбутнє сходження буде легким і простим.

Так само під час розробки команда проводить невеликий експеримент або дослідження та створює

доказ концепту (POC, Proof of Concept), який не є фактичною розробкою чи виробничими завданнями, але спрощує майбутній розвиток.

Історію (user story) не можна оцінити, поки розробницька команда не проведе дослідження обмежене по часу (не більше часу однієї ітерації). Тому спайкам не призначаються бали історії (story points), оскільки вони не пов'язані з інкрементом продукту і, отже, не сприяють швидкості команди.

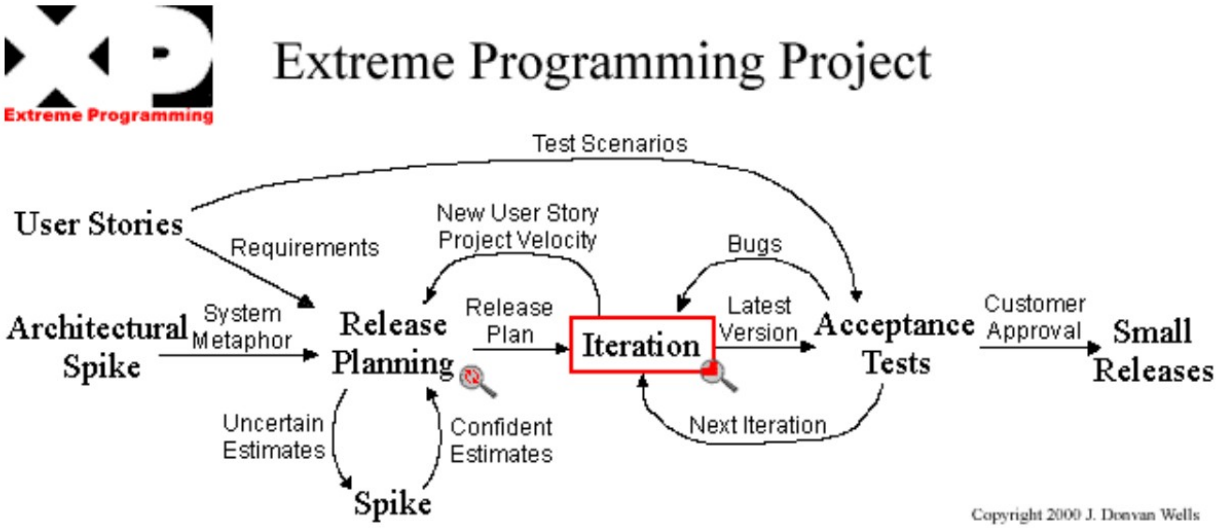
Спайки обмежені в часі.

Визначення завершеності (Definition of Done) та контроль якості (QA) в Екстремальному програмуванні (XP).

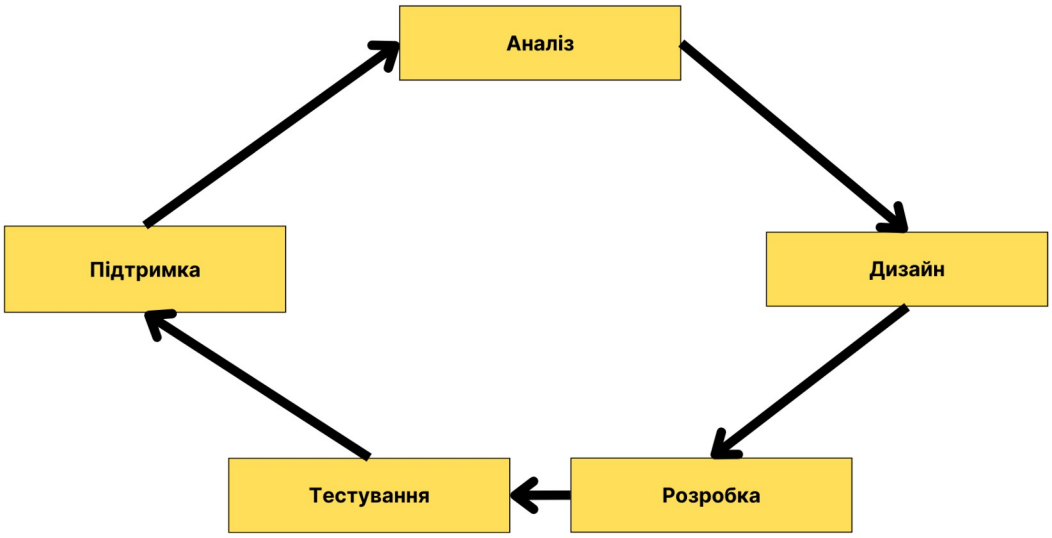
Приймальні тести (Acceptance tests) створюються під час аналізу вимог і перед кодуванням.
В Екстремальному програмуванні (XP) приймальні тести зазвичай також розробляються відповідно до методології розробки через тести (TDD).

Історія користувача (User story) не вважається завершеною, поки вона не пройде свої приймальні тести (Acceptance tests). Це означає, що нові приймальні тести повинні бути створені кожної ітерації, або команда розробників буде повідомляти про нульовий прогрес.
Контроль якості (QA) є важливою частиною процесу XP (Extreme programming).
Екстремальне програмування вимагає взаємодії розробників із QA спеціалістами.
Приймальні тести повинні бути автоматизовані, щоб їх можна було часто запускати (CI/CD).
Результати приймальних тестів публікуються команді. Це відповідальність команди запланувати час кожної ітерації для виправлення будь-яких неуспішних тестів.

Екстремальне програмування базується на TDD, автоматичних тестах.



**Цикл розробки програмного забезпечення**




Цикл розробки програмного забезпечення (ПЗ) включає такі етапи:

1. Збір та аналіз вимог,
2. Проєктування системи (дизайн),
3. Розробка (програмування),
4. Тестування,
5. Розгортання (deploy),
6. Підтримка та покращення системи (програми).

Кожен етап важливий для успішного завершення проєкту.

Зокрема, він описаний у SWEBOK від IEEE.

**Закон Літтла** описується формулою:

$L = \lambda W$

де:

L = середня кількість речей/клієнтів у системі/черзі.;

λ = середній темп прибування (інтенсивність надходження заяв);

W = середній час перебування у системі

Приклад: якщо відділення банку відвідує у середньому 10 клієнтів на годину (λ) і клієнт перебуває у ньому у середньому 1 годину (W), то середня кількість клієнтів, що знаходяться у відділені (L) досягає 10.

**Програмний продукт** = код + тести + документація (+ ліцензія + підтримка).

Наступна формула є апроксимацією, яка припускає, що всі фактори якості продукту можна наближено перевести в кількість витраченого часу на створення, налагодження та фідбек стосовно продукту.

Якість реалізації фічі (продукту) до етапу тестування обчислюється за формулою:

Якість реалізації фічі = 1 - (Час на виправлення помилок після тестування / Початкова естімація на імплементацію).

При цьому естімація враховує найкращі практики розробки, такі як оптимальність, безпека, зручність підтримки та зручність в користуванні.

Якість реалізації фічі на рівні 0.8 (до етапу QA - quality assurance) вважається нормальним показником, але вказує на потребу в додатковому часі для виправлення помилок (+20% від початкової оцінки на імплементацію).

Наприклад, якщо розробник витратив 5 днів на реалізацію фічі та 1 день на виправлення помилок, то якість реалізації фічі дорівнює 1 - 1/5, тобто 0.8.

Ідеальним варіантом є досягнення якості реалізації фічі, рівної 1, але це може бути важко досяжним на практиці через обмежені ресурси та інші фактори.

Якщо якість реалізації дорівнює 0, це означає, що час на виправлення помилок дорівнює часу на розробку.

Якість реалізації менше за 0 вказує на дуже низьку якість роботи.

Ця формула досить універсальна, щоб використовувати її для оцінки роботи як програміста та й всієї команди (якщо врахувати бета-тестування).

Історичним прикладом поганої естімації часу та вартості розробки може слугувати приклад Аналітичної машини Беббіджа, яку він так і не завершив самостійно. Також декілька мейнфреймів IBM вийшли за рамки бюджету та термінів. Однак слід зауважити, що Чарлз Беббідж був видатною постаттю у світі розробки.

Доцільно вчитися на прикладах минулого, щоб уникнути подібних помилок у майбутньому.

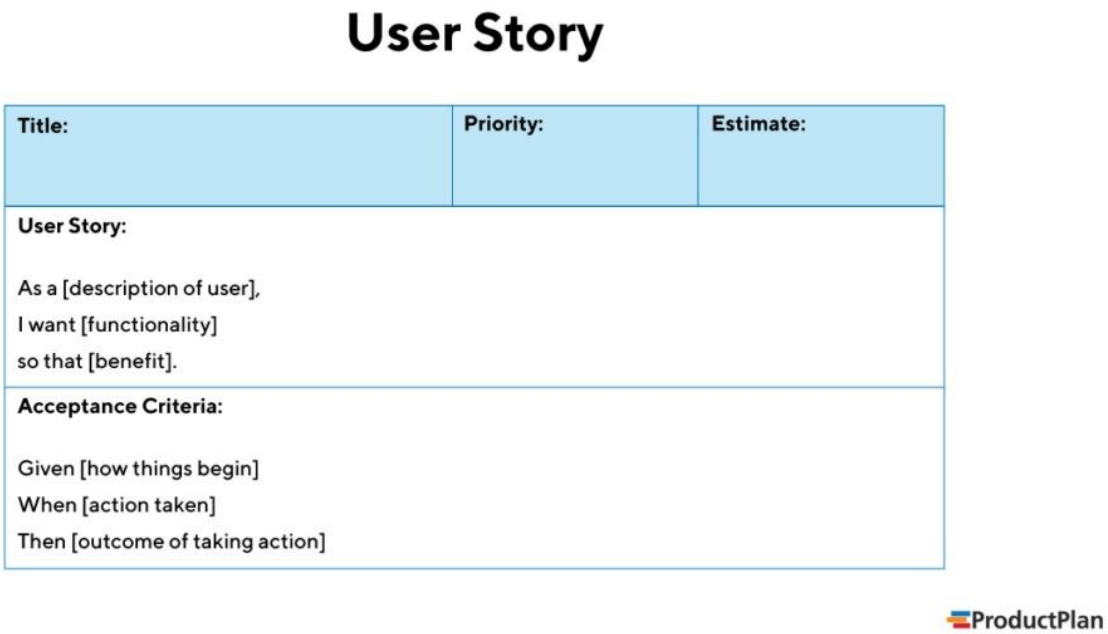
# Як писати вимоги?

Як писати вимоги Agile бізнес аналітику.

В гнучких (agile) методологіях розробки бізнес аналітик дуже близький до ролі Продукт овнера. Бізнес аналітик в гнучких методологіях пише тільки ту документацію, без якої ніяк, все інше вважається надлишковим. Definition of Done обговорюється всією Agile командою.

Абревіатура **INVEST** допомагає запам'ятати характеристики хороших історій (user stories) в Agile:

- **I – Незалежна (Independent):** Історії повинні бути максимально незалежними.
- **N – Переговорна (Negotiable):** Деталі можуть уточнюватися в процесі.
- **V – Цінна (Valuable):** Історія повинна приносити користь користувачам.
- **E – Оцінювана (Estimable):** Має бути можливість оцінити зусилля на реалізацію.
- **S – Невелика (Small):** Історія повинна бути достатньо маленькою для завершення в одному спринті.
- **T – Тестована (Testable):** Історія повинна мати чіткі критерії прийняття.




Юзер сторі (user story) - це короткий опис, не технічною мовою, того, що користувач хоче отримати.
На основі юзер сторі розробляються приймальні тести (Acceptance tests, Definition of Done).
Юзер сторі (беклог айтем), що може бути виконана в рамках спринта, вважається готовою (ready) до етапу планування спринта.

Юзер сторі та Acceptance criteria можна писати в форматі Gherkin.

---

Приклад користувацької історії (user story).

Заголовок: Дозволити користувачам скидати свої паролі

Опис: Як користувач, я хочу мати можливість скидати свій пароль, щоб я міг відновити доступ до свого облікового запису, якщо я забуду свій пароль.

Критерії прийняття:

1. Користувачі можуть запросити посилання для скидання пароля, ввівши свою зареєстровану електронну пошту.
2. Система надсилає електронний лист для скидання пароля з безпечним посиланням.
3. Користувачі можуть натиснути на посилання та ввести новий пароль.
4. Новий пароль перевіряється та зберігається.

---

Як писати вимоги бізнес аналітику:

0. Пояснити структуру вимог всім (принцип опису вимог).

1. Загальний опис зробити перед описом деталей.

2. Орієнтація на тест кейси, а не на розлогі формулювання.

3. Писати як для тупих, тобто повну інфу давати, щоб не прийшлося додумувати.
KISS - Keep it simple, stupid.

4. Підкріплювали вимоги скріншотами, якщо можливо.

5. Не писати технічні нюанси, за це відповідають розробники.

6. Вимогу перевіряє команда.

7. Не роздувати опис завдання, робити декомпозицію на декілька юзер сторі.

Перед покер плануванням юзер сторі повинна відповідати Definition of Ready, тобто бути придатною до виконання за один спринт та до планування спринту. Definition of Ready для конкретної юзер сторі визначається на зустрічі під назвою Backlog refinement. Ця зустріч не має жорстко встановлених термінів та графіку, й може відбуватися під час спринта із залученням тільки деяких членів команди, зокрема, виконавців. В Backlog refinement так чи інакше залучені Продукт овнер (та Бізнес аналітик), й можливо тім лідер. Під час рефайнменту даються загальні оцінки, щоб можна було визначити чи задача вміститься в спринт, з урахуванням team velocity. Якщо розробники не можуть оцінити задачу, береться час на дослідження (Spike).

**Детальніше:**

"Software Requirements" by Karl Wiegers and Joy Beatty,

“Professional Scrum Product Owner™ Certifications” (scrum.org).