

## Завдання

36	Замовник одна особа	Large	-	Потрібен сапорт після закінчення розробки	Video game	Приватні сервери	4 людей
----	---------------------	-------	---	---	------------	------------------	---------

№ з/п	Назви тем
1.	Відповідно до проекту визначити оптимальну методологію та життєвий цикл. Описати свій вибір
2.	Згідно описаних вимог визначити оптимальний склад команди та описати роботу кожного з членів команди
3.	Визначити та пояснити модель розробки ПЗ

4	Визначити архітектуру майбутньої системи. Пояснити вибір
5.	Визначити модель тестування та QA, пояснити вибрану модель
6.	Описати патерни проектування, які будуть використані в системі (не менше 6-8 штук, але обов'язково усі критичні)
7.	Вибрати оптимальний CI/CD підхід та пояснити його

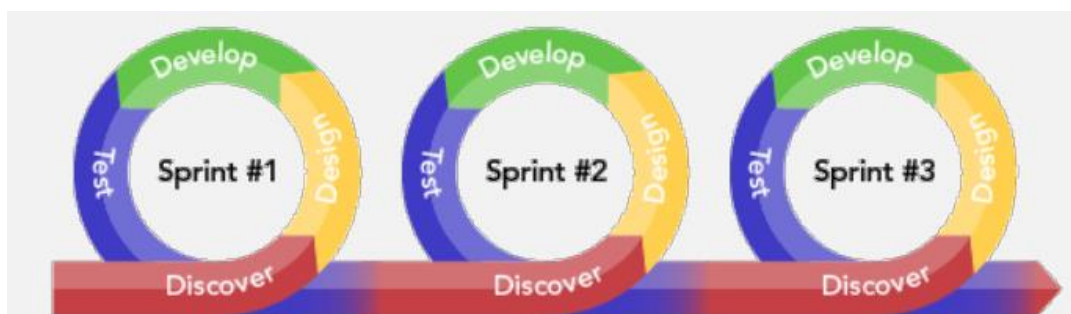
### 1. Методологія та життєвий цикл

Обираємо Agile + SCRUM через ряд причин.

Замовник один, але бюджет великий, розробляємо відео гру, час не визначений – значить вірогідно, що вимоги замовника будуть мінятися, час розтягуватися, залежно від того чи подобається йому поточний результат чи ні, чи хочуть ще щось додати, змінити, і саме тому замовнику (його представникам) треба час від часу показувати проміжні достойні результати, щоб триматись курсу, який є найкращим для клієнта і впевнюватись, що результат відповідає його очікуванням.

На початку хотілося б зробити ліричний відступ про книгу... «Кровь, пот и пиксели» Джейсона Шреєра, з якої я зрозумів, що розробка ігор - це максимально хардкордний і часто непередбачуваний процес, де в кінці-кінців всі все рівно працюють в авралі і на межі можливостей, щоб «допилити» всі фічі і зібрати все до купи.

Але теоретично, на мою думку, максимально можливий порядок може все ж таки забезпечити Agile та SCRUM власне за допомогою принципів, які лежать в їх основі:





Так як ця модель є інкрементною та ітеративною, все буде просуватись поступово і покращуватись на основі аналізу уже зробленого.

- 1) замовник та сторона компанії-розробника (та будь-які інші стейкхолдери) складають загальний список задач і побажань
- 2) створюється беклог для спринта (3 тижні), беручи частину задач із загального списку
- 3) потім дизайн, розробка, тестування і представлення проміжного результату, який потенційно може бути готовим продуктом, тобто він не розвалюється на ходу і має працювати чітко і цілісно, але в рамках запланованого (аж до готового продукту)
- 4) звіряється спринт беклог, обговорення зі стороною замовника, огляд, висновки, новий спринт

Поточно вся команда зустрічається кожного дня для обговорення, постановки задач, вирішення проблем та контролю прогресу.

Так як у нас указано, що грі потрібен іще й сапорт після здачі проекту, то відповідальність за баги і костилі теж лягає на нас, тому, крім того, щоб зараз зробити все добре, треба передбачити ресурси і на це.

## 2. Команда

З методології плавно впливає і склад команди та їхні обов'язки. Так як у нас із сторони замовника 4 людини. Одразу ж визначимо їм місце в SCRUM'і:

- 1) Product Owner, який буде представляти стейкхолдерів зі сторони замовника і в основному контролювати, чи в беклозі узгоджені інтереси замовника і практична можливість реалізації командою.

- 2) 2 стейкхолдери, які будуть говорити свої ідеї (як вони це бачать), додатково комунікувати із основним замовником і вирішувати поточні питання по загальному баченню гри. Вони можуть не знати, що має бути зроблено, вони знають, що їм потрібно.
- 3) Так як ми чесно і відкрито розробляємо нашу гру і хочемо, щоб результат був максимально хорошим, то хай остання людина з 4-ох буде тестером. Це людина зі сторони, яка точно зацікавлена в якості продукту, і також перевіряє, як воно реально працює. Зате результат буде надійним.

Далі власне іще 16 людей від нашої компанії із досить вузькими ролями, так як бюджет великий:

- 4) 1 Solution Architect – власне продумує архітектуру всієї системи, щоб все було узгоджено і враховано. Ця людина мислить на такому рівні, що розбирається у великій кількості технологій і знає, де їх і як застосовувати, бачить всю систему цілком з точки зору архітектури, чи вона буде продуманою, збалансованою і чи справді буде працювати в реальних умовах ефективно.
- 5) 1 Scrum Master – досвідчена людина, яка знає команду і контролює всю її діяльність: мітинги, прогрес, усі процеси і їх загальну взаємодію; розуміє технічні деталі проекту; комунікує і з Solution Architect'ом, і з Product Owner'ом, а також, звичайно, і з всією іншою командою, бо його задача, що всі просто ВСЕ ЗРОБИЛИ.
- 6) 1 Game Designer – геймплей, логіка та структура гри, тобто в загальному те, як буде ця «гра гратися».
- 7) 2 Художники. Займаються графікою, візуальним оформлення.
- 8) 1 Sound Designer. Відповідає за звук у грі, від найменших писків до оркестрових творів у кульмінаційні моменти. Не тільки пише музику та працює з різноманітними звуками, а відповідає за її якість у технічному плані.
- 9) 5 Програмістів. Власне кодять все, що інші напридумовували.
- 10) Іще 3 Тестери. Вони будуть перевіряти на стійкість і якість продукт, що створила команда. (разом із подвійним агентом із четвірки вище)
- 11) 2 DevOps. Інтеграція, CI/CD, робота з приватними серверами.

Всіх разом 20 людей.

### **3. Архітектура системи**

На мою думку, мікросервісна архітектура найкраще вирішить задачі даного проекту та добре доповнить вибрану методологію. На відміну від монолітної, вона не зв'язує всі компоненти воєдино, а підтримує «слабкі зв'язки», а також сприяє ітеративному та інкрементальному підходам із

отриманням на кожному кроці «potentially shippable» результату. Завдяки мікросервісам ми можемо:

- виправляти помилки, не переробляючи всю систему, а тільки займаючись конкретною її частиною;
- мати більшу свободу у використовуваних технологіях, щоб максимально підвищити продуктивність та покращити досвід користування кінцевого юзера;
- розширювати систему в тому місці і таким чином, як це потрібно для всього продукту, не прив'язуючись до інших компонентів системи, тобто «scalability».
- краще організувати роботу і чітко ділити обов'язки членів команди, концентруючи їхні зусилля на конкретних компонентах;

Тобто якщо замовник захоче після чергового спрінта щось змінити, ми зможемо працювати із окремим модульним компонентом, а якщо захоче додати – ми створимо новий компонент, який може навіть мати свою базу даних, свої якісь специфічні технології, а лише взаємодіяти з усім через певний загальний інтерфейс, бо мікросервісна система є децентралізованою. І в тому випадку, якщо гра за невизначений час існування проекту сильно розростеться, то з нею можна буде щось вдіяти, а не в один момент зрозуміти, що все в ідеалі треба було б переписати з нуля. Важливо також те, що мікросервісна архітектура добре поєднується з принципами CI/CD, що зручно в розробці.

#### 4. Тестування

Будемо проводити якомога ретельніше тестування, так як великий бюджет і є час.

Сам процес такий:

ініціація проекту > вивчення системи > складання плану тестування > створення тест кейсів > їх запуск > виявлення помилок > регресійні тести > загальний аналіз > загальний репорт

Будемо використовувати grey box тестинг, який є поєднанням black box та white box тестингу, тобто це тестування в загальному вимог та функціональності без використання внутрішнього дизайну та коду системи – black, але в критичних та проблемних місцях саме на основі коду перевіряється робота програми (statements, branches, paths, conditions і т. д.) – white.

А ось конкретні стратегії (техніки) тестування:

- unit testing (white box)  
Тестується кожен модуль індивідуально, виконується девелоперами.
- integration testing (white box)

Наступний крок після юніт текстів. Тестується взаємодія та комунікація між модулями. Так само девелопери.

- system testing (grey box)  
Тестування системи вцілому, всіх модулів у спільній взаємодії, у реальних сценаріях використання (можна сказати, наскрізне тестування);
- regression testing (grey box)  
Перевіряється те, чи виправлення помилки в одному місці не зачепить щось в іншому місці.
- performance testing (black box)  
Тестуються нефункціональні вимоги (по ідеї мають окремо визначатися), наприклад, випадок неправильних дій зі сторони користувача, невластиво великі обсяги інформації для системи, занадто швидка взаємодія і т. д.
- acceptance testing (black box)  
Кінцевий етап тестування замовником, який вирішує, чи приймати систему.

Деякі з цих видів тестування можуть бути автоматизовані.

Крім того, так як це гра, можна ще виокремити такі категорії як альфа-тестування та бета-тестування, під якими мається на увазі тестування всередині команди, що розробляє (альфа) та тестування відносно невеликою (вибраною) групою користувачів (бета).

## 5. Патерни

- Стратегія (поведінковий)  
Є набором інкапсульованих алгоритмів (стратегій), які можна замінювати один одним прямо під час роботи програми завдяки інтерфейсу (контексту).
- Спостерігач (поведінковий)  
Створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах. Тобто коли стан у спостережуваному об'єкті змінюється, то оновлювальний інтерфейс сповіщає про це всі об'єкти, в яких він визначений. У даному випадку це може використовуватись, щоб повідомляти юзеру про оновлення важливої для нього інформації.
- Одинак (породжувальний)  
Гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього. Наприклад, юзеру надаються лише визначені способи взаємодії з екземпляром, а створити нові він не може.
- Посередник (поведінковий)

Зменшити кількість класів шляхом винесення зв'язків у клас-посередник. Менше хаотичності у взаємодії між компонентами, можливість працювати із зв'язками в одному класі, що більш гнучко та зрозуміло, можливість використовувати компоненти в різних контекстах просто встановленням потрібного зв'язку.

- Адаптер (структурний)

Різні компоненти отримують змогу взаємодіяти один з одним через загальний інтерфейс. Адаптери можуть не тільки конвертувати дані з одного формату в інший, але й допомагати об'єктам із різними інтерфейсами працювати разом, що однозначно необхідно у мікросервісній архітектурі.

- Фасад (структурний)

Приховує деталі реалізації системи, дозволяючи взаємодіяти з нею за допомогою простого інтерфейсу – фасаду.

## 6. CI/CD

Використовуючи GitHub Flow створюємо дві гілки master та develop, де будемо створювати гілки з різними фічами, над якими працюємо. Спочатку комітимо зміни на гілку з фічею. Вона потрапляє в develop лише в тому випадку, якщо пройшла Continuous Integration (CI) тести і team review. Гілка з фічею може бути видалена після успішної інтеграції в develop. Будь-які commits, merges і pull requests запускають CI pipeline, що забезпечується CircleCI. Він автоматично білдить, запускає та тестує код. Merge в Master або Develop створює і реєструє контейнери, готові до деплою. Успішне проходження CI тестів на Master і Develop запускає деплоймент. Тригери CircleCI, які запускають deployment, визначені в Docker. Docker деплоїть раніше зареєстрований контейнер на self hosted сервери Amazon, окремо під development і окремо під production.

Таким чином:

- вся робота відбувається в feature гілках в develop
- кожен запушений коміт тестується CI системою
- feature гілки потрапляють в develop
- develop гілки потрапляють в master

Завдяки цьому отримана значна автоматизація процесів.

## 7. Переваги та недоліки

- Переваги
  - швидкий, організований процес;
  - надійна методологія та архітектура, де враховані ризики;
  - добре структурована команда, із чіткими сферами обов'язків;
  - гнучкість проекту до нових вимог та змін;
- Недоліки

- висока вартість (велика кваліфікована команда, кошти на CI/CD , приватні сервери);
- складність організації (відносно велика команда із значною кількістю ролей) та реалізації (мікросервісна архітектура, використання приватних серверів) проекту;
- ризики пов'язані із тим, що часові затрати на великий обсяг роботи можуть призвести до втрати актуальності проекту;