1. Słowniczek pojęć
   1. **Wymagania (requirements)** – forma pisemna w której umawiamy się z klientem, na jakiej podstawie oceni on, że nasz program działa. Opis techniczny programu w formie tabelki z wypisanymi od A do Z rzeczami które system ma robić i jak ma je robić
   2. **Historyjki użytkownika (User Stories)** – abstrakcyjny, nie techniczny opis jak program ma działać. Zbierany od klienta przez osoby techniczne jak i nie znające się na programowaniu, na ich podstawie tworzymy wymagania dla systemu
   3. **podstawa testu**: Wszystkie dokumenty, z których można wywnioskować wymagania dla modułu lub systemu. Dokumentacja, na bazie, której oparte są przypadki testowe. Jeśli dokument może być zmieniony tylko poprzez formalną procedurę zmiany, to baza testu nazywana jest zamrożoną bazą testu
   4. **Definition of Done (DoD)** – nasz opis co oznacza, że zadanie jest zrobione. Np.: „Dod: wszystkie testy przechodzą, pokrycie testami się zwiększa”. DoD można też zastosować dla całego produktu – „Kiedy produkt jest gotowy? Jak 5000 testów jest zielonych a pokrycie testami jest na poziomie 80%)
   5. **UML** – Unified Modeling Language – specjalny język wyrażający w postaci rysunków i grafów jak ma działać system (od ogólnych zagadnień do szczegółowych)
   6. **Cecha systemu** - Wyróżniający atrybut modułu lub systemu (coś co można nazwać, poddać testowaniu – np. w systemie można zmienić kolor czcionki)
   7. **BDD (Behaviour Driven Development) –** Tworzenie testów przed implementacją (część tzw. TDD – Test Driven Development) w którym opisujemy to co właśnie testujemy pisząc jak system ma się zachować. Np. „Po zalogowaniu się powinniśmy zobaczyć ekran główny programu”. Tak napisany test sprawdza faktycznie, czy zachowanie się systemu jest zgodne a nie to jak je napisaliśmy w kodzie źródłowym
   8. **ATDD (Acceptance Test Driven Development) –** również podejście “Programowanie sterowane testami - TDD” – testy generujemy z Wymagań lub historyjek użytkownika
   9. **Przypadek testowy (Test Case) –** scenariusz opisujący jeden test. Zawiera zazwyczaj:
      1. **Nazwę i numer**
      2. **Wymaganie które testujemy**
      3. **Przebieg testu krok po kroku**
      4. **Oczekiwane zachowanie**
2. Co to jest testowanie?
   1. Testowanie oprogramowania pozwala ocenić jakość oprogramowania i zmniejszyć ryzyko wystąpienia awarii podczas eksploatacji

Kom: Testując możemy powiedzieć czy coś działa i jak długo będzie działać na odpowiednim poziomie

* 1. Proces testowy składa się z
     1. Planowania (Co mamy w tym sprincie testować? Które testy będziemy pisać?)
     2. Analizy
     3. Monitorowania testów
     4. Nadzór nad testami
     5. Projektowanie i implementacja testów
     6. Raportowanie o postępie i wynikach testów
     7. Ocena jakości przedmiotu testów
  2. Testowanie dynamiczne – wymaga uruchomienia testowanego modułu lub systemu
  3. Testowanie statyczne – bez uruchamiania systemu
  4. Obejmuje analizę wymagań i historyjek użytkownika (User Stories) i kodu źródłowego
  5. Walidacja – sprawdzenie, czy system odpowiada na potrzeby użytkowników oraz innych zainteresowanych w swoim środowisku operacyjnym

1. Cele testowania
   1. Zapobieganie defektom
   2. Weryfikacja wymagań
   3. Weryfikowanie kompletności testów i ich walidacja
   4. Budowanie zaufania do poziomu jakości produktu
   5. Wykrywanie defektów i awarii i zmniejszenie poziomu ryzyka niskiej jakości produktu
   6. Dostarczenie informacji niezbędnych do świadomej poprawie wymagań systemu
   7. Przestrzeganie wymagań (również prawnych i standaryzacyjnych)
2. Czym się różni testowanie od debugowania?
   1. Testowanie – pisanie testów, uruchamianie, analiza, weryfikacja (robi je tester)
      1. Pozwala wykrywać awarie (które są skutkiem defektów)
   2. Debugowanie – proces analizy (często krok-po-kroku) działania kodu programu w celu znalezienia błędu w logice, algorytmie lub drobnych literówek. Robi je programista (i bardzo rzadko tester w niektórych zespołach)
      1. Znajdowanie, analizowanie i usuwanie defektów
   3. Testowanie potwierdzające – ma miejsce po usunięciu defektów, ma na celu sprawdzenie czy wprowadzona poprawka spowodowała usunięcie defektów
3. Dlaczego testowanie jest niezbędne?
   1. Zmiejszenie ryzyka wystąpienia awarii
   2. Wykrycie i usunięcie defektów
   3. Podniesienie jakości systemów i modułów
   4. Czasami jest wymagane w umowie lub wymagane przez prawo (np. testowanie inkubatorów)
4. Znaczenie testowania
   1. Zaangażowanie testerów już na etapie analizy wymagań pomaga wykryć absurdy, rzeczy niedoprecyzowane i trudne do testowania.
   2. Zidentyfikowanie i usunięcie defektów zmniejsza ryzyko błędnych logicznie i nietestowalnych cech oprogramowania
   3. Ścisła współpraca między testerami a projektantami systemu od początku projektu poprawia testowalność systemu
   4. Ścisła współpraca testerów z programistami pozwala lepiej zrozumieć kod i sposób jego testowania
   5. Zmniejsza ryzyko wystąpienia awarii
   6. Weryfikacja i walidacja pozwala wykryć defekty mogące prowadzić do awarii
   7. Zwiększa szanse, że program zadowoli klientów
5. Testowanie a zapewnienie jakości (przykłady)
   1. Zarządzanie jakością (Quality management) – zawiera testowanie i zapewnienie jakości (quality assurance)
   2. Zarządzanie jakością – wszystkie czynności związane z zapewnieniem jakości
   3. Zarządzanie jakością dzieli się na zapewnienie jakości i kontrola jakości
   4. Zapewnienie jakości
      1. wszystkie procesy gwarantujące nam odpowiedni (mierzalny) poziom jakości
      2. Ważna jest analiza przyczyn (root-cause analysis)
      3. Ważne spotkania scrumowe (sprint-retrospective) i wnioski z nich
   5. Kontrola jakości
      1. Szereg czynności (w tym testowych) które wspierają osiągnięcie danego (mierzalnego) poziomu jakości
6. Pomyłka – defekt – awaria
   1. Pomyłka – błąd ludzki (literówka, zmęczenie) która może prowadzić do defektu
   2. Pomyłki mogą sprawić, że defekty rozprzestrzenią się po naszych programach (ten sam programista błędnie zrobił coś w kilku miejscach)
   3. Pomyłka może zostać popełniona już w wymaganiach i prowadzić do pomyłki w kodzie
   4. Pomyłki mogą się brać z
      1. Presji czasu
      2. Zmęczenia,
      3. Braku doświadczenia
      4. Trudnego problemu do rozwiązania
      5. Nowych technologii
      6. Mało przejrzystej architektury systemu
   5. Awaria – błędne działanie systemu lub jego nagłe zamknięcie. Może się brać z defektów w kodzie jak i z defektów wynikających z warunków środowiskowych (wersja Windowsa, wersja androida, zainstalowane inne programy, antywirus itd. Itp.)
   6. Nie każdy Defekt znaleziony w kodzie oznacza awarie.
   7. Defekt – błąd w systemie wprowadzony poprzez pomyłkę. Może prowadzić do awarii.
   8. Jego analiza (root cause) pozwala zmniejszyć występowanie podobnych defektów w przyszłości
7. Siedem zasad testowania
   1. Testowanie nie wyklucza braku błędów
   2. Nie można wszystkiego przetestować
   3. Szybko znalezione błędy kosztują mniej
   4. Defekty często występują wspólnie (w podobnej formie)
   5. Powtarzanie tych samych testów nie wykrywa nowych defektów
   6. Testowanie zależy od kontekstu
   7. W systemie zawsze będą błędy
8. Wpływ kontekstu na proces testowy
   1. To jak będziemy testować zależy od wielu czynników
   2. Testujemy zgodnie z procesem tworzenia kodu (te same cykle)
   3. Na testy wpływa możliwość przeprowadzenia różnych poziomów testów
   4. Z jakim ryzykiem musimy się mierzyć
   5. Dziedzina biznesowa – czy piszemy kod dla Wojska czy biblioteki
   6. Jaki mamy budżet
   7. Dobre praktyki
      1. Mierzalne kryteria co już mamy przetestowane (test-coverage) pokrycie testami
      2. Często używa się nazwy KPI (Key Performance Indicators) jako nazwy ogólnej wszystkich sposobów w jaki mierzymy co już mamy przetestowane
9. Czynności testowe – opis i zadania
   1. Planowanie testów – definiujemy co testujemy, jaki cel chcemy osiągnąć, jakim sposobem osiągniemy cele testowania, harmonogramu testów
   2. Monitorowanie i nadzór nad testami – sprawdzamy na bieżąco czy postępujemy w stronę osiągnięcia celów testowych założonych w fazie planowania. Sprawdzamy kryteria wejścia (dane jakie dajemy do testu) i kryteria wyjścia (dane jakie otrzymujemy) pod kątem założonego zachowania systemu (lub „Definition of Done” zadania)
      1. Sprawdzamy pokrycie testami modułowymi (jednostkowymi)
      2. Oszacowanie poziomu jakości testów czarnoskrzynkowych
      3. Klienci i osoby zainteresowane otrzymują raporty o postępie testów
      4. Ustalanie priorytetów testów gdy kończy nam się czas i wiemy, że wszystkiego nie przetestujemy
      5. Dopasowywanie harmonogramu testów w zależności od tego jak ciężko jest je przeprowadzić i ile czasu trwają
      6. Dopasowywanie Kryteriów wejścia oraz DoD
   3. Analiza testów – analizujemy „co” mamy przetestować, jak będziemy to mierzyć
      1. Specyfikacja wymagań
      2. Analiza UserStories i Opowieści (epików)
      3. Analiza przypadków użycia
      4. Analizujemy tzw. Grafy przepływów sterowania (Control flow graph) – pozwala nam to na planowanie liczenia pokrycia testami
      5. Analiza diagramów UML
      6. Analiza w jakim języku zostanie zaimplementowany system, jaka będzie baza danych itd.
      7. Analiza ryzyka (jak jest szansa ze wszystko się uda)
      8. Wypisanie cech które mamy testować (coś co w systemie ma pełnić jakąś funkcję – jak to przetestować)
      9. Powiązanie realnie napisanych testów z „podstawą testów” czyli dokumentacją i danymi opisującymi system
      10. Ocena testowalności „podstawy testów” i poddanie ich analizie – co często w tego typu testowanym systemie nie działa (np. – zazwyczaj gdy wypełnimy formularz w Internecie źle – formularz jest czyszczony i musimy od nowa wpisywać dane)
   4. Projektowanie testów
      1. Tworzenie przypadków testowych (Test Cases, Test Suites)
      2. Identyfikowanie danych testowych (jakimi danymi będziemy testować system)
      3. Tworzenie środowiska testowego (przygotowanie komputerów, systemów operacyjnych i dostępów to nich dla zespołu testerów)
      4. Zestawienie środowiska do śledzenia testów i mierzenia pokrycia testami (np. Jenkins omawiany na 2 lekcji)
   5. Implementacja testów
      1. Tworzenie testów i środowiska, w którym będą one uruchomione (Tworzenie zadań w Jenkinsie i już konkretnych skryptów testowych np. w Selenium lub Pythonie)
      2. Utworzenie Zbiorów testów (Test Suites) grupujących tematycznie podobne testy
      3. Tworzenie harmonogramów testów (np. Testy Modułowe (Jednostkowe) uruchamiamy za każdym razem, a systemowe – które trwają długo raz w tygodniu w środy o północy)
      4. Przygotowanie danych testowych lub generatorów danych testowych
      5. Weryfikacja, czy testy po uruchomieniu (np. w Jenkinsie) dostarczają nam metryk i informacji o postępie testów (tzw. 2-kierunkowe śledzenie powiązań między podstawą testów, warunkami testowymi, przypadkami testowymi itd.)
   6. Wykonanie testów
      1. Uruchomienie skryptów testowych
      2. Weryfikacja testów
      3. Raportowanie defektów w narzędziach do tego przeznaczonych (Bugtracker)
      4. Uruchamianie testów regresyjnych
      5. Weryfikacja, czy testy po uruchomieniu (np. w Jenkinsie) dostarczają nam metryk i informacji o postępie testów (tzw. 2-kierunkowe śledzenie powiązań między podstawą testów, warunkami testowymi, przypadkami testowymi itd.)
   7. Ukończenie testów
      1. Raportowanie wyników testów
      2. Zebranie doświadczeń
      3. Protokół zdania systemu
10. Produkty pracy wspomagające proces testowy
11. Korzyści ze śledzenia powiązań między podstawą testów a produktami pracy związanymi z testowaniem
12. Czynniki psychologiczne wpływające na powodzenie testowania
13. Różnice w sposobie myślenia testerów i programistów