Słowniczek:

**IoT (Internet of things)** – nowe pojęcie mające na celu połączenie w sieć urządzeń na świecie (np. mogę włącznikiem u siebie w domu włączyć lampkę nocną w Australii, wykonywać zdalnie operacje w szpitalu itd.)

**Cykl życia oprogramowania (LifeCycle)** – gdy budujemy program (lub samochód) często robimy to w powtarzalnych krokach na różnych poziomach tworzenia samochodu. Robiąc silnik możemy wyróżnić te same kolejne etapy pracy jak podczas robienia podwozia (planowanie – tworzenie – testowanie integracyjne – testowanie systemowe – itd.). Cykl życia jest dość ogólnym pojęciem i opisuje konkretny etap na którym tworzymy (lub testujemy )daną część systemu/samochodu.

**Logika biznesowa –** Jest to zchowanie systemu które spełnia swoje komercyjne/biznesowe założenie. Np. logiką biznesową silnika jest to, że napędza osie, i sprawia, że auto się porusza. Cała reszta to tylko składowe elementy (tłoki, zbiornik paliwa itd.) które służa temu, by silnik spełniał swoją logikę biznesową

1）Modele cyklu życia oprogramowania

Kandydat potrafi wyjaśnić relacje między czynnościami związanymi

z wytwarzaniem oprogramowania a czynnościami testowymi w cyklu życia

oprogramowania.

Kandydat potrafi wskazać powody, dla których konieczne jest dostosowanie

modeli cyklu życia oprogramowania do kontekstu wynikającego z charakterystyki

projektu i produktu.

1. Wytwarzanie oprogramowania a testowanie oprogramowania
   1. Dla każdego etapu tworzenia oprogramowania (budowania samochodu) tworzony jest etap testów (etapy te mogą się składać z różnych poziomów testów w zależności od tego jak dużą część testujemy – np. test integracyjny silnika, test systemowy silnika, a później znów test integracyjny auta i systemowy auta)
   2. Analizę i projektowanie testów należy zacząć podczas tworzenia danego elementu (testy silnika analizujemy i zaczynamy równolegle z inżynierami budującymi ten silnik)
   3. Testerzy powinni uczestniczyć w procesie tworzenia wymagań od samego początku razem z inżynierami (programistami)
   4. W sylabusie dzielimy sposób wytwarzania oprogramowania (Cykle życia) na 2 grupy
   5. Sekwencyjne (jak waterfall każdy etap planowanie – budowa – testowanie następuje po zakończeniu poprzedniego)
   6. Iteracyjne i przyrostowe – czyli pracujemy w sprintach i co iteracja (sprint) dostarczamy nowy działający kawałek systemu
   7. Przyrostowy sposób może mieć kilka form (RUP, SCRUM, Kanban, Spirala – sylabus str. 35 do wkucia różnic)
   8. Zaleta przyrostowego sposobu – bardzo szybko klient otrzymuje część systemu i szybko może dać nam informację zwrotną czy mu się on podoba
2. Modele cyklu życia oprogramowania w kontekście
   1. To jakie cykle życia wybierzemy dla naszego projektu – zależy od kontekstu (jak wszystko zresztą). Należy wybrać taki model który najlepiej odpowiada naszemu produktowi (samochodowi) i pominąć niepotrzebne procesy. Np. samochód musi mieć dobre hamulce i ten etap możemy podzielić na więcej kroków testowych bo jest kluczowy dla bezpieczeństwa pasażera i norm dopuszczenia do jazdy
   2. To dla kogo robimy produkt, jak będziemy go sprzedawać ( w salonach, półkach sklepowych), jak długo będziemy go sprzedawać (np. samochód przez 2-3 lata a później wejdzie nowszy model) – to wszystko ma wpływ na to jaki sposób projektu wybierzemy i jakie będzie miał cykle

2）. Poziomy testów

Kandydat potrafi porównać poszczególne poziomy testów z punktu widzenia

celów, podstawy testów, przedmiotów testów, typowych defektów i awarii, podejść

i odpowiedzialności.

1. W sylabusie wyróżniamy
   1. Modułowe (jednostkowe, UT)
   2. Intergracyjne (testujemy 2 modułe ze sobą)
   3. Systemowe (jak działa system względem wymagań)
   4. Akceptacyjne (wykonuje je klient sprawdzając czy system spełnia jego oczekiwania, my też możemy u siebie je odtworzyć by mieć większą pewność że klient zaakceptuje nasz produkt)
2. Każdy z powyższych etapów testów ma następujące właściwości
   1. Ma swoje cele (jednostkowe – pokrycie kodu, integracyjne – sprawdzanie czy 2 moduły działają ze sobą, itd.)
   2. Ma swoją podstawę testów (jednostkowe – kod źródłowy, integracyjne – scenariusze testowe, systemowe – wymagania klienta itd.)
   3. Przedmiot testów (co właśnie testujemy)
   4. Typowe dla swojego etapu defekty i awarie
   5. Podejścia i odpowiedzialności
3. Każdy etap testów wymaga trochę innej strategii i technologii
4. Testowanie modułowe :
   1. Technika białoskrzynkowa
   2. Skupia się na działaniu kodu źródłowego
   3. Zalety – zmniejszenie ryzyka, wykrywanie defektów w kodzie
   4. Wychwytywanie błędów które spowodowały by, że testy wyższego poziomu – np. integracyjne – na pewno by się nie udały
   5. W iteracyjnych sposobach wytwarzania oprogramowania – wykonuje się je ciągle, co cykl, co zmiana (np. w Jenkinsie)
   6. Pisząc te testy izolujemy moduł od jego środowiska i testujemy go samego (np. testując silnik – wyjmujemy go, zamiast prawdziwych kabli i połączeń wkładamy zaślepki (Mock Object) które tylko udają pracę w prawdziwym środowisku)
   7. Podstawa testów: projekt, architektura kodu, kod, model danych, specyfikacje modułów
   8. Testujemy moduły (w programowaniu nazywane też klasami) i ogólnie kod źródłowy (dlatego są to testy białoskrzynkowe)
   9. Wykonuje je programista (w 99% przypadków). Często testowanie jest wykorzystywane jako sposób na tworzenie testów zanim zaczniemy pisać kod (TDD) – np. tworzymy stacje testującą silnik w naszej stacji kontroli pojazdów – zanim w ogóle zaczniemy ten silnik budować
5. Testowanie integracyjne
   1. Testujemy połączenie dwóch komponentów
   2. Cele testowania integracyjnego:
      1. Zmiejszenie ryzyka (jak każdego😊 etapu)
      2. Sprawdzenie czy zachowania funkcjonalne (czy 2 komponenty razem wykonują pracę jaką mają – tłok – napęd) i niefunkcjonalne (czy ich połączenie jest użyteczne, wydajne itd.)
      3. Budowanie zaufania dla interfejsów (interfejs to „połączenie” 2 modułów – np. radio łączy się z wtykiem radia w desce rozdzielczej. Testujemy, czy „interfejs” naszego radia (gniazdko) ma dobry interfejs (dobrze działa z wtyczką, i ewentualnie np. z wtyczkami w innych samochodach)
      4. Znajdowanie defektów
      5. Zapobieganie „wydostawania się” błędów do testów wyższego poziomu
   3. Testy integracyjne można często automatyzować (np. klikanie na stronach internetowym narzędziem selenium symulującym klikającego użytkownika)
   4. Skupia się na połączeniach pomiędzy komponentami
   5. Podstawą testów są:
      1. Projekty systemu (np. rysunech techniczny połączenia tłoka z napędem)
      2. Diagramy sekwencji (specjalne diagramy w UML)
      3. Specyfikacje protokołów (np. specyfikacja gniazdka prądu – jego napięcia i izolacja)
      4. Przypadki użycia (UML)
      5. Architektura systemu (silnika)
   6. Testujemy integracyjnie (zazwyczaj) połączenia:
      1. Z bazą danych
      2. Z bibliotekami
      3. Z innymi komponentami (poprzez ich interfejsy – lub inaczej API)
      4. Połączenia z serwisami (np. serwis pogody)
   7. Typowe defekty wykrywane:
      1. System nie radzi sobie z błędnymi danymi
      2. Dane przychodzą w złym czasie(za wolno, za szybko)
      3. Błędy w dopasowaniu interfejsów
      4. Błąd w nawiązaniu połączenia między modułami
      5. Źle obsługiwane błędy
      6. Złe założenia projektowe które wychodzą dopiero teraz (np. ktoś przygotował bezpiecznik w mA a źródło prądu w kA)
   8. Odpowiedzialności testów integracyjnych:
      1. Testujemy głównie połączenia (interfejsy) modułów
      2. Modułów nie testujemy oddzielnie tylko ich współpracę ze sobą
      3. Wykonują je głównie testerzy
6. Testy systemowe
   1. Kolejny – wyższy poziom testów
   2. Testujemy zachowanie całego „systemu” (np. silnika)
   3. Cele:
      1. Redukcja ryzyka (ale niespodzianka😊)
      2. Walidacja czy system wykonuje zakładaną pracę
      3. Tworzenie zaufania do systemu
      4. Odnajdowanie defektów
      5. Zapobieganie wydostania się defektów na wyższe poziomy
   4. Basa testów
      1. Specyfikacje całego systemu
      2. Analiza ryzyka
      3. Przypadki użycia (UML)
      4. Modele zachowania systemu
      5. Historyjki użytkownika
   5. Testujemy tak:
      1. Całe aplikacje (np. cały samochów)
      2. Elementy softwarowe i hardware’owe
      3. Instrukcje obsługi (np. czy auto zachowuje się tak jak napisaliśmy w instrukcji)
7. Testy akceptacyjne
   1. Podobnie jak Systemowe skupiają się na zachowaniu całego systemu
   2. Cele:
      1. Ustalenie ogólnej jakości systemu
      2. Walidacja pracy jaką wykonuje system
      3. Weryfikowanie funkcjonalnych i niefunkcjonalnych zachowań systemu
   3. Często dostarczają informacji jak blisko „gotowości systemu” jesteśmy
   4. Może zawierać testy alpha i beta
   5. UAT – User Acceptance Testing – testy skupiające się na zachowaniu systemu pod względem użytkownika (czy auto jeździ)
   6. OAT – Operational Acceptance Testing – testy czy administracja samochodem działa (czy da się podpiąć komputer pokładowy itd. Itp.)
   7. Szukanie słabych punktów systemu (np. czy któraś część szybko się psuje )
   8. Typowe testy OAT zawierają:
      1. Kopia zapasowa danych
      2. Powrót do działania po awarii
      3. Dokumentację
      4. Wydajność
      5. Bezpieczeństwo i regulacje systemu

3）. Typy testów

Kandydat potrafi porównać testowanie funkcjonalne z testowaniem

niefunkcjonalnym i białoskrzynkowym. Kandydat zdaje sobie sprawę z tego, że testy funkcjonalne, niefunkcjonalne i białoskrzynkowe mogą występować na dowolnym poziomie testowania. Kandydat potrafi porównać przeznaczenie testowania potwierdzającego i testowania regresji.

1. Testy funkcjonalne (testujemy jak system się zachowuje!)
2. Testy niefunkcjonalne (testujemy czy system ma wady nie związane z jego uruchomieniem – wygląd, szerokość, ostre zderzaki, zły rozstaw kół itp.)
3. Testy białej skrzynki (White box testing)
   1. Testowanie widząc „zawartość” czy też „bebechy” tego co testujemy (np. testy jednostkowe)
   2. Najczęściej możemy dla takich testów dostarczyć informacji o „pokryciu” kodu testami
4. Testy czarnej skrzynki (Black Box testing) – testujemy jakiś zamknięty system do którego mamy dostęp tylko poprzez jego interfejsy
5. Testowanie zmian
   1. Często chodzi tu o testy regresji (cały zbiór dotychczasowych testów uruchamiany w całości po każdej zmianie)
   2. Gdy w systemie zachodzi zmiana (tłoki nowej firmy bo stare się nie sprawdzały) należy przetestować wpływ tej zmiany na dotychczas działający system
   3. Testy potwierdzające – gdy w systemie znaleziony zostaje defekt i jest on naprawiony – należy to potwierdzić pisząc nowy test
6. Poziomy testów a typy testów
   1. Każdy poziom testów ma przedstawiciela z każdego rodzaju testów (jednostkowe, integracyjne, systemowe)
   2. Przykłady s sylabusa:
      1. N potrzeby testowania modułowego projektowane są testy odzwierciedlające sposób, w jakidany moduł powinien obliczać odsetki składane
      2. na potrzeby testowania integracji modułów projektowane są testy odzwierciedlające sposób, w jaki informacje na temat konta pozyskane poprzez interfejs użytkownika są przekazywane do warstwy logiki biznesowej
      3. na potrzeby testowania systemowego projektowane są testy odzwierciedlające sposób, w jaki posiadacze rachunków mogą składać wnioski o przyznanie linii kredytowej na rachunku bieżącym
      4. na potrzeby testowania integracji systemów projektowane są testy odzwierciedlające sposób, w jaki dany system sprawdza zdolność kredytową posiadacza rachunku przy użyciu zewnętrznej mikrousługi
      5. na potrzeby testowania akceptacyjnego projektowane są testy odzwierciedlające sposób w jaki pracownik banku zatwierdza lub odrzuca wniosek kredytowy
   3. Testy niefunkcjonalne – przykłady
      1. na potrzeby testowania modułowego projektowane są testy wydajnościowe, które umożliwiają ocenę liczby cykli procesora niezbędnych do wykonania złożonych obliczeń dotyczących łącznej kwoty odsetek
      2. na potrzeby testowania integracji modułów projektowane są testy zabezpieczeń, które mają na celu wykrycie słabych punktów zabezpieczeń związanych z przepełnieniem bufora danymi przekazywanymi z interfejsu użytkownika do warstwy logiki biznesowej
      3. na potrzeby testowania systemowego projektowane są testy przenaszalności, które umożliwiają sprawdzenie, czy warstwa prezentacji działa we wszystkich obsługiwanych przeglądarkach i urządzeniach przenośnych
      4. na potrzeby testowania integracji systemów projektowane są testy niezawodności, które pozwalają na dokonanie oceny odporności systemu w przypadku braku odpowiedzi ze strony mikrousługi służącej do sprawdzania zdolności kredytowej
      5. na potrzeby testowania akceptacyjnego projektowane są testy użyteczności, które pozwalają ocenić ułatwienia dostępu dla osób niepełnosprawnych zastosowane w interfejsie do przetwarzania kredytów po stronie banku

4） Testowanie pielęgnacyjne (są z tego 3-4 pytania nawet!)

Kandydat potrafi wymienić i omówić zdarzenia wyzwalające testowanie

pielęgnacyjne. Kandydat potrafi opisać rolę analizy wpływu w testowaniu pielęgnacyjnym.

1. Gdy już wprowadzimy samochód na rynek (rok go budowaliśmy testowaliśmy wszystko się udało, auto zaczyna jeździć po drogach) jako firma która go dostarczyła teraz „pielęgnujemy” nasz produkt na rynku. Na pewno coś przeoczyliśmy i teraz klienci używając naszego produktu zaczną zgłaszać usterki. My poprawiamy te usterki i testujemy nasze poprawki – to nazywa się utrzymaniem produktu (Maintenance) lub pielęgnacją
2. Po każdej zmianie w fazie pielęgnacji trzeba wykonać testowanie pielęgnacyjne, które sprawdza czy nasza poprawka zadziałała oraz czy nic po drodze nie zepsuła we wcześniejszym działaniu systemu
3. Pielęgnacja często jest dużo bardziej ostrożna i zawiera więcej kroków (w końcu poprawiamy coś co weryfikować będzie nasz klient)
4. Zdarzenia wywołujące pielęgnację:
   1. Modyfikacja – coś udoskonaliliśmy w systemie, naprawiliśmy błąd, zmieniliśmy system operacyjny, wprowadziliśmy aktualizację jakiejś biblioteki
   2. Migracja – gdy zmieniamy platformy (np. przechodzimy z naszym samochodem na rynek brytyjski i kierownica będzie z drugiej strony). Prędkościomierz pokazuje teraz Mile / h zamiast kilometrów.
   3. Wycofanie – projekt samochód dobiega końca – chcemy go wycofać z rynku, ale chcemy też zachować jego projekt, dane itd. – to też powoduje testy pielęgnacyjne (testowanie procedur odzyskiwania i pozyskiwania zarchiwizowanych danych)
   4. Gdy testujemy coś z Internetu rzeczy (IoT – Internet of things) – testujemy pielęgnacyjnie po każdej poprawce (np. nowe złącze żarówki w naszej zdalnej lampie 😊)
5. Analiza wpływu związana z integracją – patrzymy jaki wpływ na nasz produkt (jego jakość, bezpieczeństwo, wygląd) ma nasza zmiana
   1. Jeśli nie przestrzegamy naszego Quality Assurance procesu – czyli zapewnienie że wszystkie procedury się odbywają – możemy uniemożliwić dobrą analizę zmian – bo stracimy np. informację co już działało w przeszłości albo dlaczego działało (ktoś wprowadził coś do silnika ale tego nie udokumentował – teraz nie wiemy czemu działa i czy dalej działa)