1.1 Co to jest testowanie

Rozdział 1.1 "Co to jest testowanie?" wprowadza podstawowe informacje o testowaniu oprogramowania. Wyjaśnia, że testowanie to proces oceny oprogramowania w celu znalezienia błędów, upewnienia się, że spełnia wymagania i działa zgodnie z oczekiwaniami. Testowanie pomaga również identyfikować ryzyko związane z błędami, które mogą wpłynąć na użytkowników lub organizację.

Podkreślono, że testowanie jest niezbędne, ponieważ błędy mogą występować na każdym etapie tworzenia oprogramowania – od specyfikacji wymagań po implementację i eksploatację. Dzięki testowaniu można zmniejszyć negatywne skutki tych błędów, takie jak niezadowolenie klientów, straty finansowe czy uszkodzenie reputacji.

Rozdział tłumaczy także, że testowanie nie jest czynnością jednorazową, lecz procesem ciągłym, który towarzyszy rozwojowi oprogramowania, od planowania po wdrożenie.

1.1.1 Cele testów

Rozdział 1.1.1 "Cele testów" wyjaśnia, dlaczego testowanie jest kluczowym elementem tworzenia i utrzymania oprogramowania. Podstawowe cele testów to:

1. **Znajdowanie defektów** – Testowanie pomaga zidentyfikować błędy w oprogramowaniu, zanim trafi ono do użytkowników.
2. **Zapewnienie jakości** – Dzięki testom można upewnić się, że oprogramowanie działa zgodnie z wymaganiami i oczekiwaniami.
3. **Ocena zgodności z wymaganiami** – Testy potwierdzają, czy system spełnia określone wymagania funkcjonalne i niefunkcjonalne.
4. **Zapobieganie defektom** – Proces testowania może pomóc unikać błędów w przyszłości, np. dzięki wnioskom z poprzednich projektów.
5. **Minimalizowanie ryzyka** – Testy identyfikują potencjalne problemy, które mogłyby wpłynąć na użytkowników, finanse lub reputację.
6. **Budowanie zaufania** – Regularne i rzetelne testowanie zwiększa pewność, że oprogramowanie jest stabilne i gotowe do użycia.

Podsumowując, cele testów to nie tylko znajdowanie błędów, ale także budowanie pewności, że produkt działa poprawnie i spełnia potrzeby użytkowników.

[?]W kontekście rozdziału 1.1.1 sylabusa ISTQB, ***historyjki użytkownika (user stories)*** *to krótkie opisy funkcjonalności oprogramowania z perspektywy użytkownika. Są one często używane w metodykach zwinnych (np. Scrum) i pomagają jasno określić, co system ma robić, aby spełnić potrzeby użytkownika.*

Każda historyjka użytkownika zwykle składa się z trzech elementów:

1. ***Rola użytkownika** – Kto korzysta z funkcji? (np. "Jako klient sklepu internetowego...")*
2. ***Cel** – Co użytkownik chce osiągnąć? (np. "...chcę dodać produkt do koszyka...")*
3. ***Korzyść** – Dlaczego to jest ważne? (np. "...żeby móc go później kupić.")*

W rozdziale tym wskazano, że testowanie może być powiązane z historyjkami użytkownika poprzez tworzenie testów akceptacyjnych. Te testy weryfikują, czy funkcjonalność spełnia wymagania

opisane w historyjce i dostarcza wartość użytkownikowi. Dzięki temu historyjki pomagają w zrozumieniu, jakie cele mają testy, i pozwalają lepiej ukierunkować ich tworzenie.

1.1.2 Testowanie a debugowanie

Rozdział 1.1.2 "Testowanie a debugowanie" wyjaśnia różnice między tymi dwoma procesami, które często są mylone:

Testowanie

- To proces wykonywania oprogramowania w celu znalezienia błędów i oceny jego jakości.
- Testerzy poszukują symptomów problemów, takich jak nieprawidłowe wyniki, błędy w działaniu lub odstępstwa od wymagań.
- Głównym celem testowania jest identyfikacja defektów, ale testerzy nie naprawiają ich bezpośrednio.

Debugowanie

- To proces naprawy błędów znalezionych w oprogramowaniu.
- Debugowanie jest zwykle przeprowadzane przez programistów, którzy analizują kod, identyfikują przyczynę problemu i wprowadzają poprawki.
- Często po debugowaniu przeprowadza się ponowne testy, aby upewnić się, że problem został skutecznie rozwiązany.

Kluczowa różnica:

- **Testowanie** wykrywa problem, ale nie rozwiązuje go.
- **Debugowanie** zajmuje się analizą i usunięciem przyczyny problemu.

Podsumowując, testowanie i debugowanie to różne, ale wzajemnie uzupełniające się procesy w cyklu tworzenia oprogramowania.

1.2 Dlaczego testowanie jest niezbędne?

Rozdział 1.2 "Dlaczego testowanie jest niezbędne?" podkreśla, że testowanie odgrywa kluczową rolę w zapewnianiu jakości oprogramowania i minimalizowaniu ryzyka związanego z jego użytkowaniem. Oto najważniejsze powody:

1. Błędy są nieuniknione

- Ludzie, którzy tworzą oprogramowanie, mogą popełniać błędy na różnych etapach – od planowania, przez projektowanie, po implementację.
- Nawet doświadczeni programiści nie są wolni od pomyłek.

2. Konsekwencje błędów mogą być poważne

- Błędy w oprogramowaniu mogą prowadzić do:
    - Strat finansowych (np. błędne obliczenia, awarie systemów bankowych).
    - Uszczerbku na reputacji firmy (np. awarie aplikacji publicznych).
    - Zagrożenia dla życia i zdrowia (np. błędy w systemach medycznych czy motoryzacyjnych).

3. Testowanie pomaga zidentyfikować problemy wcześniej

- Im wcześniej zostanie znaleziony błąd, tym łatwiej i taniej jest go naprawić.
- Znalezienie defektów na wczesnym etapie zmniejsza ryzyko wystąpienia krytycznych problemów w fazie produkcyjnej.

4. Zapewnienie zgodności z wymaganiami

- Testowanie pozwala upewnić się, że system spełnia oczekiwania użytkowników i wymagania biznesowe.

5. Minimalizacja ryzyka

- Regularne testy pomagają zidentyfikować i zminimalizować ryzyko związane z awariami, opóźnieniami w dostawach lub niezadowoleniem użytkowników.

6. Budowanie zaufania

- Skuteczne testowanie zwiększa pewność, że system działa poprawnie, jest stabilny i gotowy do użycia przez klientów.

Podsumowując, testowanie jest niezbędne, ponieważ zapewnia jakość, zmniejsza ryzyko i pomaga uniknąć kosztownych błędów, które mogą zaszkodzić firmie, użytkownikom i społeczeństwu.

1.2.2. Testowanie a zapewnienie jakości

Rozdział 1.2.2 "Testowanie a zapewnienie jakości" wyjaśnia różnicę między testowaniem a szerszym pojęciem zapewnienia jakości (QA, ang. *Quality Assurance*), jednocześnie pokazując, jak te dwa elementy się uzupełniają.

Zapewnienie jakości (QA)

- Jest to całościowe podejście mające na celu zapobieganie błędom na wszystkich etapach cyklu życia oprogramowania.
- Obejmuje procesy, polityki, standardy i działania, które mają na celu dostarczenie wysokiej jakości produktu.
- Skupia się na **zapobieganiu problemom** i budowaniu odpowiednich procedur od samego początku (np. projektowanie zgodne z wymaganiami, przeglądy kodu).

Testowanie

- Testowanie jest jednym z elementów QA, skoncentrowanym na **identyfikowaniu defektów** poprzez sprawdzanie, czy produkt działa zgodnie z oczekiwaniami.
- Testowanie jest reakcją na potencjalne problemy – przeprowadza się je, aby sprawdzić, czy defekty zostały wprowadzone w procesie tworzenia oprogramowania.

Kluczowe różnice

- **QA to działania zapobiegawcze** – koncentrują się na poprawie procesu, aby unikać błędów w przyszłości.
- **Testowanie to działania detekcyjne** – koncentrują się na wykrywaniu błędów już istniejących w oprogramowaniu.

Jak się uzupełniają?

- QA dba o to, aby procesy wytwarzania oprogramowania były zoptymalizowane i minimalizowały ryzyko defektów.
- Testowanie dostarcza informacji zwrotnej, czy te procesy rzeczywiście działają skutecznie, poprzez wykrywanie problemów w produkcie końcowym.

Podsumowanie

Zapewnienie jakości i testowanie to dwa różne, ale współpracujące ze sobą elementy, które mają wspólny cel – dostarczenie wysokiej jakości oprogramowania. QA dba o proces, a testowanie sprawdza rezultat tego procesu.

1.2.3. Pomyłki, defekty, awarie i podstawowe przyczyny

Rozdział 1.2.3 "Pomyłki, defekty, awarie i podstawowe przyczyny" wyjaśnia, jak błędy w oprogramowaniu powstają i jakie mają konsekwencje. Oto najważniejsze pojęcia:

1. Pomyłki (*mistakes*, zwane też błędami ludzkimi)

- Są to działania lub decyzje człowieka, które prowadzą do wprowadzenia defektu w oprogramowaniu.
- Przykład: programista źle zrozumiał wymagania albo pomylił się podczas pisania kodu.

2. Defekty (*defects*, zwane też błędami lub bugami)

- To niedoskonałości w kodzie, dokumentacji, procesie lub wymaganiach, które mogą prowadzić do nieprawidłowego działania systemu.
- Defekty powstają na skutek pomyłek człowieka.
- Przykład: nieprawidłowa formuła w algorytmie lub brak walidacji danych wejściowych.

3. Awarie (*failures*)

- To rzeczywiste problemy w działaniu systemu, które występują, gdy defekt zostanie aktywowany w trakcie jego używania.
- Przykład: użytkownik wpisuje niepoprawne dane, system nie działa zgodnie z oczekiwaniami (np. aplikacja się zawiesza).

Nie wszystkie defekty prowadzą do awarii – niektóre mogą pozostać niewykryte, jeśli nie zostaną aktywowane w określonych warunkach.

Podstawowe przyczyny defektów i awarii

1. **Ludzkie błędy i ograniczenia**
    - Niewłaściwe zrozumienie wymagań, złożoność systemu, presja czasu lub niedostateczne przeszkolenie zespołu.
2. **Problemy w komunikacji**
    - Niewłaściwe przekazywanie wymagań między interesariuszami, programistami, analitykami i testerami.
3. **Złożoność technologii**

    - Nowe narzędzia, systemy czy technologie mogą prowadzić do pomyłek z powodu ich trudności w obsłudze.
4. **Zmiany w wymaganiach**
    - Częste zmiany lub niejasne wymagania mogą powodować powstawanie defektów.
5. **Środowisko i warunki użytkowania**
    - Różnice między środowiskiem testowym a rzeczywistym mogą ujawniać defekty w niespodziewanych okolicznościach.

Podsumowanie

Pomyłki ludzi prowadzą do powstawania defektów, a te z kolei mogą skutkować awariami, gdy zostaną aktywowane. Zrozumienie tych zależności pomaga minimalizować ryzyko i poprawiać procesy tworzenia oraz testowania oprogramowania.

1.3. Zasady testowania

Rozdział 1.3 "Zasady testowania" przedstawia siedem podstawowych zasad, które pomagają skutecznie testować oprogramowanie. Oto one:

1. Testowanie ujawnia obecność defektów, a nie ich brak

- Testowanie może wykazać, że w systemie występują defekty, ale nigdy nie dowiedzie, że defektów brak.
- Nawet przy intensywnym testowaniu nie można zagwarantować, że system jest całkowicie wolny od błędów.

2. Testowanie wyczerpujące jest niemożliwe

- Nie da się przetestować wszystkich możliwych kombinacji danych wejściowych, ścieżek czy warunków.
- Testowanie musi być priorytetyzowane i skoncentrowane na najważniejszych obszarach.

3. Wczesne testowanie oszczędza czas i pieniądze

- Im wcześniej defekty zostaną wykryte (np. w fazie wymagań lub projektowania), tym mniej kosztuje ich naprawa.
- Zasada ta wspiera ideę wczesnego zaangażowania testowania w cykl życia oprogramowania (ang. *shift-left testing*).

4. Zgrupowanie defektów (zasada Pareto)

- Większość defektów (np. 80%) jest często spowodowana przez niewielką liczbę modułów lub funkcji (np. 20%).
- Testowanie powinno być skoncentrowane na tych „ryzykownych” obszarach systemu.

5. Paradoks pestycydów

- Jeśli te same testy są powtarzane wielokrotnie, przestają być skuteczne w wykrywaniu nowych defektów.
- Testy muszą być regularnie przeglądane i aktualizowane, aby pozostały skuteczne.

6. Testowanie zależy od kontekstu

- Podejście do testowania zależy od rodzaju i celu oprogramowania.
    - Systemy krytyczne (np. medyczne) wymagają bardziej rygorystycznego testowania.
    - Gry komputerowe mogą mieć inne priorytety, np. wrażenia użytkownika.

7. Brak błędów to iluzja

- Nawet jeśli system jest wolny od defektów, może nie spełniać potrzeb użytkowników lub wymagań biznesowych.
- Testowanie powinno być ukierunkowane nie tylko na brak błędów, ale także na zgodność z oczekiwaniami i wartością dla użytkownika.

Podsumowanie

Te zasady są fundamentem skutecznego testowania. Ich stosowanie pozwala optymalizować procesy testowe, minimalizować ryzyko i zapewniać wyższą jakość oprogramowania.

*[?]**Shift-left testing** to podejście, które polega na przesunięciu testowania na wcześniejsze etapy cyklu życia oprogramowania (stąd „shift-left" – przesunięcie w lewo na osi czasu). Tradycyjnie testowanie zaczynało się po zakończeniu fazy implementacji, ale shift-left promuje włączenie testowania już od początkowych etapów, takich jak analiza wymagań czy projektowanie.*

1.4. Czynności testowe, testalia i role związane z testami

Rozdział 1.4 "Czynności testowe, testalia i role związane z testami" wprowadza podstawowe informacje o procesie testowania, artefaktach powstających w jego trakcie oraz rolach osób zaangażowanych w testowanie.

1.4.1. Czynności i zadania testowe

Proces testowania składa się z kilku kluczowych etapów, które są realizowane w określonej kolejności:

1. **Planowanie testów**
    - Określenie celów testowania, zasobów, harmonogramu i strategii.
    - Zdefiniowanie, co będzie testowane i jak.
2. **Analiza testów**
    - Analizowanie wymagań, specyfikacji i innych źródeł informacji, aby zidentyfikować co należy przetestować.
    - Tworzenie listy funkcji, przypadków użycia czy ryzyk do przetestowania.
3. **Projektowanie testów**
    - Opracowanie szczegółowych przypadków testowych na podstawie analizy.
    - Przygotowanie danych testowych oraz środowiska testowego.
4. **Implementacja testów**
    - Przygotowanie testów do uruchomienia: zautomatyzowanie przypadków testowych (jeśli to możliwe) lub skonfigurowanie testów manualnych.
5. **Wykonanie testów**
    - Uruchamianie przypadków testowych, zbieranie wyników i rejestrowanie defektów.

- Monitorowanie i raportowanie postępów.

6. **Ewaluacja wyników testów**
    - Ocena, czy wszystkie wymagania zostały spełnione.
    - Weryfikacja, czy produkt jest gotowy do przekazania użytkownikom.
7. **Zakończenie testów**
    - Archiwizowanie dokumentacji testowej i danych.
    - Podsumowanie wyników i wniosków z testów, co może posłużyć w przyszłych projektach.

1.4.2. Proces testowy w kontekście

Rozdział ten wyjaśnia, że proces testowy można dostosować do specyficznego kontekstu projektu. Różne rodzaje projektów i środowisk wymagają innych podejść, priorytetów i technik testowania. Wartościowe testowanie zależy od uwzględnienia kontekstu, w którym oprogramowanie powstaje i będzie używane.

Czynniki wpływające na proces testowy

1. **Typ systemu lub aplikacji**
    - **Systemy krytyczne dla bezpieczeństwa** (np. medyczne, lotnicze) wymagają bardziej rygorystycznych testów i przestrzegania norm.
    - **Aplikacje konsumenckie** (np. aplikacje mobilne) mogą skupiać się na użyteczności i wydajności.
2. **Cel testowania**
    - Czy głównym celem jest wykrycie defektów, weryfikacja zgodności z wymaganiami, czy może poprawa użyteczności?
3. **Rodzaj projektu**
    - Projekty **agile** (zwinne) wymagają dynamicznych i częstych testów, z silnym naciskiem na automatyzację.
    - Projekty **waterfall** (kaskadowe) mogą wymagać bardziej formalnych, dokumentowanych testów.
4. **Dostępne zasoby i ograniczenia**
    - **Budżet i czas:** Większe ograniczenia wymagają priorytetyzacji testów na najważniejszych obszarach.
    - **Zasoby ludzkie i narzędzia:** Wpływają na wybór podejścia – manualnego, automatycznego, czy mieszanki obu.
5. **Ryzyko**
    - Testowanie powinno być skoncentrowane na obszarach o najwyższym ryzyku, takich jak krytyczne funkcje systemu lub podatności bezpieczeństwa.
6. **Doświadczenie i kompetencje zespołu**
    - Zespół z doświadczeniem w automatyzacji może efektywniej wykorzystać zaawansowane narzędzia, natomiast zespół mniej zaawansowany może polegać bardziej na testach manualnych.

Dostosowanie procesu testowego

Proces testowy musi być dostosowany do kontekstu, obejmując m.in.:

- **Wybór technik testowych:** W zależności od wymagań i rodzaju systemu (np. testy eksploracyjne, testy regresji, testy bezpieczeństwa).
- **Stopień formalności:** Projekty krytyczne wymagają ścisłej dokumentacji testowej, natomiast w zwinnych procesach może ona być minimalna.
- **Automatyzacja:** Testy automatyczne są kluczowe w przypadku częstych zmian w kodzie (np. w projektach agile).
- **Kultura organizacyjna:** Współpraca między zespołami, otwartość na testowanie i rola testerów w projekcie mogą się różnić w zależności od organizacji.

Podsumowanie

Proces testowy w kontekście oznacza, że testowanie nie ma jednego uniwersalnego podejścia – powinno być dopasowane do specyficznych wymagań projektu, jego celów, ograniczeń i ryzyk. Kluczowe jest elastyczne podejście, które uwzględnia zarówno wymagania techniczne, jak i biznesowe oraz ograniczenia zespołu i projektu.

1.4.3. Testalia

Rozdział ten wyjaśnia, czym są testalia oraz jakie artefakty powstają w trakcie procesu testowego. Testalia to wszystkie dokumenty, dane, narzędzia i inne artefakty używane lub tworzone podczas testowania. Mają kluczowe znaczenie dla skutecznego zarządzania i realizacji testów.

Rodzaje testaliów

1. **Planowanie testów:**
    - **Plan testów:** Dokument określający strategię, zakres, cele, zasoby i harmonogram testów.
    - **Harmonogram testów:** Szczegółowy terminarz działań testowych.
2. **Przygotowanie i analiza:**
    - **Specyfikacja przypadków testowych:** Zestaw scenariuszy testowych opisujących kroki do wykonania oraz oczekiwane wyniki.
    - **Dane testowe:** Wartości wejściowe, które są używane podczas wykonywania testów.
3. **Wykonanie i rejestracja:**
    - **Raporty defektów:** Opis znalezionych błędów, ich priorytety, kroki do ich reprodukcji i statusy.
    - **Logi testów:** Automatyczne lub manualne zapisy wyników wykonanych testów.
4. **Ocena i raportowanie:**
    - **Raporty z testów:** Podsumowanie wyników testów, przetestowanego zakresu, znalezionych defektów i rekomendacji.
    - **Metryki testowe:** Dane o efektywności testowania (np. liczba przypadków testowych, liczba wykrytych defektów).
5. **Narzędzia i inne zasoby:**

- **Narzędzia testowe:** Automatyzujące testy (np. Selenium), wspierające zarządzanie testami (np. JIRA, TestRail) czy monitorujące środowisko testowe.
- **Środowisko testowe:** Konfiguracja systemów, baz danych, serwerów, symulatorów używanych w testach.

Charakterystyka testaliów

- **Precyzyjność i kompletność:** Dobrze przygotowane testalia minimalizują ryzyko błędów w testowaniu i ułatwiają odtwarzanie scenariuszy testowych.
- **Śledzenie i powtarzalność:** Dokumenty i narzędzia umożliwiają odtworzenie wyników testów oraz śledzenie błędów na każdym etapie.
- **Automatyzacja:** Wspiera szybkość i dokładność procesu testowego poprzez użycie skryptów i narzędzi automatyzujących.

Znaczenie testaliów w procesie testowym

1. **Efektywność:**
    - Ułatwiają organizację pracy testerów i zarządzanie procesem testowym.
2. **Jakość:**
    - Pomagają zapewnić, że testowanie pokrywa wszystkie kluczowe funkcjonalności i ryzyka.
3. **Współpraca:**
    - Dobrze przygotowane testalia wspierają komunikację między członkami zespołu (testerami, programistami, kierownikami).
4. **Dowód jakości:**
    - Dokumenty, takie jak raporty z testów, są formalnym potwierdzeniem przeprowadzonych działań testowych, co jest istotne w projektach zewnętrznych lub audytach.

Podsumowanie

Testalia to fundament dobrze zorganizowanego procesu testowego. Ich staranne przygotowanie, użycie i przechowywanie wspierają jakość testowania, pomagają w śledzeniu postępów i defektów oraz są dowodem na rzetelne wykonanie pracy w zakresie zapewniania jakości.

Rozdział ten opisuje, jak łączyć podstawę testów (np. wymagania, specyfikacje) z testaliami (przypadki testowe, raporty). Dzięki temu można upewnić się, że wszystkie wymagania zostały przetestowane i niczego nie pominięto.

Dlaczego to ważne?

- **Ocena pokrycia:** Czy każdy wymóg ma przypisany test?
- **Śledzenie zmian:** Łatwo zaktualizować testy po zmianie wymagań.
- **Przejrzystość:** Pokazuje interesariuszom, co zostało przetestowane.

Jak to zrobić?

- **Identyfikatory:** Każdy wymóg i test mają unikalny numer.
- **Macierz śledzenia:** Tabela łącząca wymagania z przypadkami testowymi i wynikami.

- **Narzędzia:** Programy, np. JIRA, automatyzują powiązania.

Korzyści

- Łatwiejsze zarządzanie testami.
- Pewność, że system spełnia wszystkie wymagania.
- Szybsze reagowanie na zmiany w projekcie.

Podsumowując, śledzenie powiązań zapewnia, że testy są kompletne i dobrze zorganizowane.

1.4.5. Role w procesie testowania

W procesie testowania różne osoby pełnią określone role, które wspólnie zapewniają wysoką jakość produktu. Każda rola wnosi unikalne umiejętności i odpowiada za inne aspekty testowania.

Kluczowe role

1. **Tester:**
    - Wykonuje testy zgodnie z planem.
    - Zgłasza znalezione defekty i monitoruje ich naprawę.
    - Przygotowuje dane testowe i raporty z wyników testów.
2. **Kierownik testów:**
    - Planowanie i organizacja całego procesu testowego.
    - Zarządzanie zasobami, harmonogramem i ryzykiem.
    - Monitorowanie postępów i raportowanie do interesariuszy.
3. **Analityk testowy:**
    - Analiza wymagań i przygotowanie szczegółowych przypadków testowych.
    - Identyfikacja ryzyk i priorytetów testowych.
4. **Automatyk testów:**
    - Tworzy i utrzymuje skrypty automatyzujące testy.
    - Dba o wydajność i stabilność narzędzi do automatyzacji.
5. **Programista:**
    - Współpracuje z testerami w naprawie defektów.
    - Czasami wykonuje testy jednostkowe i integracyjne.
6. **Właściciel produktu / interesariusz:**
    - Definiuje wymagania biznesowe i zatwierdza wyniki testów.
    - Określa priorytety funkcjonalności i ryzyk.

Dlaczego podział ról jest ważny?

- **Skuteczność:** Każda osoba koncentruje się na swojej specjalizacji.
- **Współpraca:** Role ułatwiają koordynację między zespołami.
- **Jakość:** Zróżnicowane perspektywy pomagają lepiej zidentyfikować problemy i spełnić wymagania.

Podsumowując, jasno określone role w procesie testowym pozwalają efektywnie planować, przeprowadzać i monitorować testy, zapewniając wysoki poziom jakości produktu.

1.5.1. Ogólne umiejętności wymagane w związku z testowaniem

Rozdział ten opisuje podstawowe umiejętności i cechy, które są niezbędne do efektywnego wykonywania testowania. Testerzy muszą wykazywać zarówno kompetencje techniczne, jak i miękkie, aby skutecznie wspierać proces zapewniania jakości.

Umiejętności techniczne

1. **Znajomość procesu testowego:**
    - Rozumienie etapów testowania, takich jak planowanie, projektowanie przypadków testowych, wykonanie i raportowanie.
2. **Znajomość technik testowania:**
    - Stosowanie technik projektowania testów, takich jak ekwiwalencja klas, analiza wartości granicznych, tablice decyzyjne itp.
3. **Umiejętność analizy:**
    - Czytanie i zrozumienie wymagań, specyfikacji i podstawy testów w celu identyfikacji obszarów wymagających testowania.
4. **Znajomość narzędzi testowych:**
    - Obsługa narzędzi do zarządzania testami (np. JIRA, TestRail) i automatyzacji (np. Selenium, Postman).
5. **Podstawy programowania:**
    - Przydatne w testach automatycznych, białoskrzynkowych lub analizie kodu.
6. **Znajomość środowiska testowego:**
    - Konfiguracja środowisk i urządzeń niezbędnych do przeprowadzania testów.

Umiejętności miękkie

1. **Dokładność i skrupulatność:**
    - Zwracanie uwagi na szczegóły podczas analizy wyników i raportowania defektów.
2. **Komunikacja:**
    - Jasne przekazywanie wyników testów, problemów i sugestii interesariuszom, programistom i kierownikom.
3. **Rozwiązywanie problemów:**
    - Identyfikowanie przyczyn defektów i współpraca z zespołem w celu ich eliminacji.
4. **Krytyczne myślenie:**
    - Podejście analityczne do testowanego systemu w celu znalezienia potencjalnych problemów, które mogą być trudne do zauważenia.
5. **Współpraca:**
    - Praca zespołowa w różnych kontekstach, np. w modelu zwinnych zespołów Agile lub tradycyjnych modelach kaskadowych.
6. **Adaptacyjność:**

- Szybkie dostosowywanie się do zmian w wymaganiach, harmonogramach lub narzędziach.

Znaczenie umiejętności ogólnych

Testerzy muszą łączyć umiejętności techniczne z kompetencjami miękkimi, aby skutecznie identyfikować problemy i wspierać proces zapewnienia jakości. Równowaga między tymi cechami pozwala testerom efektywnie współpracować w zespołach i zapewniać wysoką jakość produktów.

Podsumowanie

Testerzy powinni być wszechstronni – rozumieć techniczne aspekty procesu testowego, ale także potrafić pracować w grupie, jasno komunikować wyniki i elastycznie reagować na zmieniające się warunki projektu.

1.5.2. Podejście „cały zespół”

Ten rozdział podkreśla znaczenie podejścia *„cały zespół”* (ang. *Whole-Team Approach*) w procesie testowania. Zakłada ono, że odpowiedzialność za jakość oprogramowania spoczywa na całym zespole, a nie tylko na testerach.

Kluczowe założenia podejścia „cały zespół”

1. **Wspólna odpowiedzialność za jakość:**
    - Każdy członek zespołu (tester, programista, analityk biznesowy, właściciel produktu) ma wpływ na jakość końcowego produktu.
    - Testowanie jest integralną częścią całego cyklu wytwarzania oprogramowania, a nie jedynie ostatnim etapem.
2. **Współpraca i komunikacja:**
    - Zespół wspólnie analizuje wymagania, identyfikuje ryzyka i planuje testy.
    - Regularna wymiana informacji między rolami zapewnia lepsze zrozumienie produktu i szybsze wykrywanie problemów.
3. **Integracja testowania w procesie wytwórczym:**
    - Testowanie odbywa się na każdym etapie – od planowania po wdrożenie (np. poprzez testy jednostkowe, integracyjne, akceptacyjne).
    - W metodach zwinnych (Agile) testerzy współpracują z programistami przy tworzeniu testów automatycznych i specyfikacji testów.

Korzyści podejścia „cały zespół”

1. **Lepsza jakość produktu:**
    - Współpraca między rolami minimalizuje ryzyko przeoczenia defektów.
    - Wspólna odpowiedzialność za jakość zwiększa zaangażowanie zespołu.
2. **Szybsze wykrywanie problemów:**
    - Wczesne zaangażowanie testerów w analizę wymagań pozwala wychwycić błędy już na etapie planowania.
3. **Większa efektywność:**

- Testowanie wplecione w codzienną pracę zespołu eliminuje niepotrzebne opóźnienia.

4. **Lepsza komunikacja:**
    - Regularne spotkania zespołowe (np. w Agile – daily stand-ups) zapewniają wymianę wiedzy i synchronizację działań.

Praktyczne zastosowanie

- W Agile i DevOps podejście *„cały zespół”* jest standardem.
- Zespół wspólnie tworzy strategie testowe, planuje testy akceptacyjne, a programiści i testerzy współpracują nad automatyzacją testów.

Podsumowanie

Podejście *„cały zespół”* promuje współodpowiedzialność za jakość, bliską współpracę i wczesne zaangażowanie testerów. Dzięki temu proces testowania staje się bardziej efektywny, a produkt – lepszy i bardziej dopasowany do oczekiwań użytkowników.

1.5.3. Niezależność testowania

Rozdział ten opisuje, czym jest niezależność testowania, jakie są jej poziomy oraz dlaczego warto ją stosować w procesie testowania oprogramowania.

Czym jest niezależność testowania?

- Niezależność testowania oznacza, że osoby przeprowadzające testy nie są bezpośrednio zaangażowane w tworzenie testowanego oprogramowania.
- Dzięki temu zmniejsza się ryzyko stronniczości i przeoczenia błędów wynikających z „ślepoty na własne błędy”.

Poziomy niezależności

1. **Brak niezależności:**
    - Testy są przeprowadzane przez programistów, którzy napisali testowany kod.
    - Ryzyko: Mogą przeoczyć własne błędy z powodu zaangażowania w proces tworzenia.
2. **Niezależność wewnętrzna:**
    - Testy są wykonywane przez osoby z tego samego zespołu, ale nie przez autora kodu.
    - Zaleta: Lepsza obiektywność w porównaniu do braku niezależności.
3. **Niezależność zewnętrzna:**
    - Testy są wykonywane przez niezależny dział testowy lub zewnętrzny zespół.
    - Zaleta: Wysoka obiektywność i możliwość spojrzenia na produkt z perspektywy użytkownika.
    - Ryzyko: Może wystąpić brak pełnego zrozumienia kontekstu biznesowego.
4. **Niezależność użytkownika:**
    - Testy są przeprowadzane przez docelowych użytkowników (np. testy akceptacyjne).
    - Zaleta: Testowanie oparte na rzeczywistych potrzebach i oczekiwaniach.

Korzyści niezależności testowania

1. **Obiektywność:** Testerzy niezależni mogą dostrzec defekty, które autorzy kodu mogą przeoczyć.
2. **Lepsza jakość testów:** Zewnętrzne spojrzenie pozwala lepiej ocenić produkt z perspektywy użytkownika.
3. **Zwiększenie zaufania:** Niezależne testy budują większe zaufanie do wyników testowania.

Ryzyka związane z niezależnością testowania

1. **Komunikacja:** Zespół niezależny może mieć ograniczoną wiedzę na temat kontekstu biznesowego lub technicznego projektu.
2. **Opóźnienia:** Włączenie niezależnego zespołu testowego może wydłużyć czas realizacji projektu, jeśli komunikacja nie jest odpowiednio zorganizowana.
3. **Koszty:** Niezależność testowania, zwłaszcza zewnętrzna, często generuje dodatkowe koszty.

Zastosowanie w praktyce

- W małych projektach testy są zazwyczaj wykonywane przez programistów lub testerów wewnętrznych.
- W większych projektach i systemach krytycznych stosuje się zewnętrzne zespoły testowe lub audyt testowy.

Podsumowanie

Niezależność testowania zwiększa obiektywność i skuteczność testów, ale wymaga odpowiedniego balansu między kosztami, wiedzą o projekcie i potrzebami biznesowymi. Idealny poziom niezależności zależy od charakteru projektu, ryzyk i dostępnych zasobów.

2.1. Testowanie w kontekście modelu cyklu wytwarzania oprogramowania

Rozdział 2.1 opisuje, jak testowanie jest zintegrowane z różnymi modelami cyklu życia oprogramowania. Podkreśla, że testowanie powinno być dostosowane do konkretnego modelu oraz potrzeb projektu.

Cykl wytwarzania oprogramowania a testowanie

Testowanie jest częścią całego cyklu wytwarzania oprogramowania. Jego miejsce i zakres zależą od modelu cyklu życia:

1. **Modele sekwencyjne (np. kaskadowy):**
    - Każda faza projektu następuje po zakończeniu poprzedniej.
    - Testowanie jest często skoncentrowane na późniejszych etapach (np. po fazie implementacji).
    - **Ryzyko:** Wykrycie defektów późno w cyklu zwiększa koszty ich naprawy.
2. **Modele iteracyjne i przyrostowe (np. Agile):**
    - Rozwój i testowanie odbywają się w krótkich cyklach (iteracjach).
    - Testowanie jest zintegrowane z procesem wytwarzania na każdym etapie.
    - **Korzyść:** Wczesne wykrywanie defektów oraz możliwość szybkiego reagowania na zmiany.

Poziomy testowania w cyklu życia oprogramowania

1. **Testy jednostkowe:**
    - Przeprowadzane przez programistów.
    - Koncentrują się na małych fragmentach kodu, takich jak funkcje czy moduły.
2. **Testy integracyjne:**
    - Sprawdzają współdziałanie pomiędzy modułami lub systemami.
    - Wykonywane przez programistów i/lub testerów.
3. **Testy systemowe:**
    - Sprawdzają całość systemu w kontekście wymagań.
    - Wykonywane przez testerów.
4. **Testy akceptacyjne:**
    - Oceniają system z perspektywy użytkownika końcowego lub interesariuszy.
    - Weryfikują, czy produkt spełnia wymagania biznesowe.

Testowanie a modele cyklu życia

1. **Model kaskadowy:**
    - Testowanie zaczyna się po zakończeniu fazy implementacji.
    - Wynikiem takiego podejścia może być późne wykrywanie defektów.
    - Zalecane: włączenie wczesnych przeglądów i statycznych technik testowych, aby zmniejszyć ryzyko.
2. **Model V:**
    - Testowanie jest planowane równolegle do fazy tworzenia oprogramowania.
    - Każda faza rozwoju (np. projektowanie) ma przypisaną fazę testowania (np. testy systemowe).
    - Sprzyja lepszej organizacji testów w całym cyklu.
3. **Modele iteracyjne (np. Agile):**
    - Testowanie jest częścią każdej iteracji.
    - Testy są realizowane równolegle z rozwojem oprogramowania.
    - Wymaga automatyzacji i ciągłej współpracy testerów, programistów i właściciela produktu.

Rola wczesnego testowania

- Wczesne zaangażowanie testowania w projekt pozwala:
    - Wykrywać i usuwać defekty już na etapie analizy wymagań lub projektowania.
    - Zmniejszać koszty naprawy defektów w późniejszych fazach.
    - Lepiej planować strategię testową.

Podsumowanie

Testowanie powinno być dostosowane do modelu cyklu życia oprogramowania. W modelach iteracyjnych testowanie jest integralną częścią całego procesu, podczas gdy w modelach

sekwencyjnych wymaga szczególnego uwzględnienia wczesnych przeglądów i strategii testowych, aby ograniczyć ryzyka i koszty.

2.1.2. Model cyklu wytwarzania oprogramowania a dobre praktyki testowania

Ten podrozdział podkreśla znaczenie stosowania dobrych praktyk testowania w kontekście różnych modeli cyklu wytwarzania oprogramowania. Zawiera kluczowe wytyczne dotyczące organizacji testów, które pomagają zapewnić skuteczność i efektywność w całym procesie tworzenia oprogramowania.

Dobre praktyki testowania w modelu cyklu życia

1. **Wczesne zaangażowanie testowania:**
    - Testowanie powinno rozpoczynać się jak najwcześniej w cyklu wytwarzania oprogramowania (np. podczas analizy wymagań lub projektowania).
    - Wczesne przeglądy dokumentacji i statyczne techniki testowania pozwalają na szybkie wykrycie defektów.
    - Korzyść: Niższe koszty naprawy błędów i lepsza jakość końcowa.
2. **Integracja testów w proces:**
    - Testowanie powinno być ściśle zintegrowane z każdym etapem cyklu życia, niezależnie od modelu (np. kaskadowy, iteracyjny).
    - W modelach zwinnych (Agile) testy są częścią każdej iteracji, a testerzy aktywnie współpracują z programistami i interesariuszami.
3. **Zarządzanie ryzykiem:**
    - Identyfikacja i ocena ryzyk (np. związanych z funkcjonalnością, bezpieczeństwem) powinna kierować priorytetami testów.
    - Testowanie kluczowych funkcji i obszarów wysokiego ryzyka powinno mieć pierwszeństwo.
4. **Automatyzacja testów:**
    - Automatyzacja powtarzalnych testów (np. regresji) zwiększa efektywność i umożliwia częste testowanie w krótkich cyklach.
    - Szczególnie ważna w modelach iteracyjnych i DevOps.
5. **Przejrzysta dokumentacja:**
    - Tworzenie jasnych i zrozumiałych planów testów, przypadków testowych i raportów zapewnia lepszą komunikację między zespołami.
6. **Weryfikacja i walidacja:**
    - Testy powinny obejmować zarówno weryfikację (czy produkt jest zgodny ze specyfikacją), jak i walidację (czy spełnia oczekiwania użytkownika).
7. **Ciągłe doskonalenie:**
    - Regularne retrospektywy i analiza wyników testów pomagają ulepszać proces testowy.

Dobre praktyki w różnych modelach cyklu życia

1. **Model kaskadowy:**

    - Szczegółowe planowanie testów na wczesnych etapach.
    - Przeprowadzanie przeglądów dokumentacji (np. wymagań, projektów) jako formy wczesnego testowania.
2. **Model V:**
    - Ścisłe powiązanie faz testowania z fazami rozwoju (np. testy jednostkowe po kodowaniu, testy systemowe po projektowaniu systemu).
    - Zastosowanie testowania weryfikacyjnego i walidacyjnego na każdym poziomie.
3. **Modele iteracyjne i Agile:**
    - Codzienna współpraca testerów, programistów i interesariuszy.
    - Wykorzystanie automatyzacji testów regresyjnych i ciągłej integracji.
    - Częste testowanie w każdej iteracji (ang. *continuous testing*).

Podsumowanie

Dobre praktyki testowania, takie jak wczesne zaangażowanie, zarządzanie ryzykiem, automatyzacja i ciągłe doskonalenie, są kluczowe dla sukcesu projektu. Ich zastosowanie powinno być dostosowane do specyfiki wybranego modelu cyklu życia oprogramowania, aby zapewnić skuteczność testów i wysoką jakość produktu końcowego.

2.1.3. Testowanie jako czynnik określający sposób wytwarzania oprogramowania

Te metody wpływają na proces wytwarzania oprogramowania, ponieważ kształtują sposób, w jaki testowanie i rozwój są ze sobą zintegrowane.

TDD (Test-Driven Development)

- **Opis:** Testowanie sterowane rozwojem to podejście, w którym programista najpierw pisze testy jednostkowe, zanim stworzy implementację funkcji, które te testy mają sprawdzić.
- **Cel:**
    - Upewnienie się, że kod jest testowalny od samego początku.
    - Zapewnienie, że funkcje są tworzone zgodnie z wymogami testów.
- **Wpływ na proces:**
    - Promuje iteracyjne podejście do programowania.
    - Testy są pisane równolegle z kodem, co minimalizuje ryzyko zapomnienia o krytycznych przypadkach testowych.

ATDD (Acceptance Test-Driven Development)

- **Opis:** Rozwój sterowany testami akceptacyjnymi skupia się na pisaniu testów akceptacyjnych przed implementacją funkcji. Testy te są tworzone przy współpracy zespołu, w tym programistów, testerów i właścicieli produktu.
- **Cel:**
    - Upewnienie się, że system spełnia wymagania biznesowe i oczekiwania interesariuszy.
    - Zmniejszenie ryzyka nieporozumień dotyczących wymagań.
- **Wpływ na proces:**
    - Zwiększa zrozumienie wymagań między zespołami.
    - Testy akceptacyjne stają się częścią dokumentacji, co ułatwia ich utrzymanie.

BDD (Behavior-Driven Development)

- **Opis:** Rozwój sterowany zachowaniem kładzie nacisk na definiowanie funkcjonalności w formie prostych, zrozumiałych scenariuszy testowych, które opisują, jak system powinien się zachowywać. Scenariusze te są zwykle pisane w języku naturalnym (np. za pomocą formatu **Given-When-Then**).
- **Cel:**
    - Ułatwienie komunikacji między członkami zespołu, w tym nietechnicznymi interesariuszami.
    - Skupienie na zachowaniu systemu z perspektywy użytkownika końcowego.
- **Wpływ na proces:**
    - Sprzyja tworzeniu bardziej przejrzystego i łatwego do zrozumienia kodu.
    - Łączy specyfikację funkcji z testami, eliminując ryzyko rozbieżności.

Zastosowanie w praktyce

- **Agile i DevOps:**
    - Te podejścia są szczególnie popularne w metodykach zwinnych i DevOps, ponieważ sprzyjają iteracyjnemu rozwojowi i częstym testom.
- **Automatyzacja testów:**
    - Zarówno ATDD, jak i BDD często wymagają automatyzacji testów, co integruje proces testowy z wytwarzaniem oprogramowania.

Podsumowanie

Metody TDD, ATDD i BDD są przykładem, jak testowanie może kształtować sposób tworzenia oprogramowania, integrując wymagania biznesowe, techniczne i użytkownika w procesie rozwoju. Dzięki nim testowanie staje się kluczowym elementem planowania i realizacji projektu, a nie tylko jego końcowym etapem.

2.1.4. Metodyka DevOps a testowanie

Metodyka DevOps integruje rozwój oprogramowania (**Dev**) i operacje (**Ops**) w celu przyspieszenia dostarczania oprogramowania, poprawy jego jakości i zwiększenia współpracy między zespołami. W tym kontekście testowanie odgrywa kluczową rolę, aby zapewnić ciągłą jakość produktu w dynamicznym i szybkim cyklu rozwoju.

Rola testowania w DevOps

1. **Ciągłe testowanie (Continuous Testing):**
    - Testy są wykonywane na każdym etapie cyklu życia oprogramowania, od integracji kodu, przez budowę, aż po wdrożenie.
    - Automatyzacja testów jest kluczowym elementem, aby nadążyć za szybkim tempem DevOps.
2. **Integracja z ciągłą integracją i ciągłym dostarczaniem (CI/CD):**
    - Testowanie jest wbudowane w potoki CI/CD, gdzie kod jest automatycznie testowany po każdej zmianie lub integracji.

    - Testy regresyjne, funkcjonalne, bezpieczeństwa i wydajności są wykonywane automatycznie, zanim kod zostanie wdrożony na środowisko produkcyjne.
3. **Automatyzacja testów:**
    - W DevOps automatyzacja jest kluczowa, szczególnie dla testów regresyjnych, integracyjnych i akceptacyjnych.
    - Narzędzia automatyzujące pomagają w szybkim wykrywaniu błędów i ich naprawie.
4. **Testowanie w środowiskach produkcyjnych:**
    - DevOps zakłada, że pewne rodzaje testów, takie jak monitorowanie zachowania systemu, testy wydajności czy testy eksploracyjne, mogą być wykonywane bezpośrednio w środowisku produkcyjnym.
    - W tym celu stosuje się m.in. testowanie "canary releases" (wdrożenia tylko dla części użytkowników) lub techniki "dark launches" (udostępnienie funkcji w tle).
5. **Shift-left i Shift-right testing:**
    - **Shift-left testing:** Akcent na testowanie we wczesnych etapach procesu wytwarzania (np. analiza wymagań, testy jednostkowe).
    - **Shift-right testing:** Testowanie w późniejszych etapach, w tym monitorowanie, testy wydajności i regresyjne w środowisku produkcyjnym.

Korzyści testowania w DevOps

1. **Wysoka jakość oprogramowania:**
    - Dzięki ciągłemu testowaniu i automatyzacji zmniejsza się ryzyko defektów w kodzie produkcyjnym.
2. **Szybkie wykrywanie błędów:**
    - Testy wbudowane w CI/CD pozwalają na natychmiastowe wykrycie i naprawę problemów.
3. **Szybsze dostarczanie:**
    - Zintegrowane testowanie umożliwia szybsze wprowadzanie zmian i aktualizacji do produkcji.
4. **Zwiększenie współpracy:**
    - Testowanie w DevOps wymaga ścisłej współpracy między programistami, testerami i zespołami operacyjnymi.

Wyzwania testowania w DevOps

1. **Automatyzacja:**
    - Wymaga odpowiednich narzędzi, czasu i wiedzy, aby stworzyć i utrzymać stabilne testy automatyczne.
2. **Testowanie w dynamicznych środowiskach:**
    - Środowiska chmurowe i konteneryzacja (np. Docker, Kubernetes) wymagają elastycznego podejścia do testowania.
3. **Zarządzanie danymi testowymi:**

- Generowanie i zarządzanie danymi testowymi, które są reprezentatywne i bezpieczne (np. zgodne z RODO), może być wyzwaniem.

Podsumowanie

Testowanie w DevOps to kluczowy element umożliwiający dostarczanie wysokiej jakości oprogramowania w szybkim i ciągłym cyklu. Automatyzacja, ciągłe testowanie oraz integracja z CI/CD pozwalają na efektywne zarządzanie jakością, ale wymagają także odpowiedniego planowania, narzędzi i umiejętności.

2.1.5. Przesunięcie w lewo (ang. *shift left approach*)

Przesunięcie w lewo to podejście, które polega na przesunięciu działań związanych z testowaniem na wcześniejsze etapy cyklu życia oprogramowania. Celem jest identyfikacja i eliminacja problemów tak wcześnie, jak to możliwe, aby zmniejszyć koszty ich naprawy i poprawić ogólną jakość produktu.

Kluczowe aspekty Shift-Left Approach

1. **Wczesne zaangażowanie testowania:**
    - Testerzy uczestniczą w działaniach od samego początku projektu, takich jak analiza wymagań, planowanie i projektowanie.
    - Pozwala to na wcześniejsze wykrycie nieścisłości w wymaganiach i projektach.
2. **Zapobieganie defektom zamiast ich wykrywania:**
    - Dzięki analizie wymagań i projektów można zapobiegać defektom, zanim zostaną zakodowane.
    - To podejście zmniejsza ryzyko poważnych błędów w późniejszych etapach.
3. **Współpraca zespołowa:**
    - Metody Shift-Left promują bliską współpracę między programistami, testerami i interesariuszami biznesowymi, aby zapewnić lepsze zrozumienie wymagań i celów projektu.
4. **Automatyzacja testów:**
    - Wczesne testy jednostkowe i integracyjne są często zautomatyzowane, co umożliwia szybsze iteracje.
    - Automatyzacja jest kluczowa w modelach zwinnych (Agile) i DevOps, gdzie szybkie dostarczanie oprogramowania jest priorytetem.

Przykłady działań w ramach Shift-Left

1. **Przeglądy wymagań i projektów:**
    - Testerzy uczestniczą w przeglądach dokumentacji wymagań i projektów technicznych, aby identyfikować potencjalne problemy.
2. **Test-Driven Development (TDD):**
    - Testy jednostkowe są pisane przed implementacją kodu, co zapewnia, że funkcje są testowalne od samego początku.
3. **Automatyzacja testów już na etapie kodowania:**

- Tworzenie testów jednostkowych i integracyjnych w czasie pisania kodu.

4. **Prototypowanie i wczesne testy użyteczności:**
    - Weryfikacja interfejsów i funkcjonalności z udziałem użytkowników końcowych na wczesnym etapie projektu.

Korzyści przesunięcia w lewo

1. **Redukcja kosztów:**
    - Wykrycie defektów w początkowych fazach projektu jest mniej kosztowne niż ich naprawa po wdrożeniu.
2. **Wyższa jakość produktu:**
    - Dzięki szybszemu wykrywaniu i eliminacji defektów produkt końcowy jest bardziej niezawodny.
3. **Szybszy rozwój:**
    - Unikanie poważnych błędów w późnych fazach przyspiesza cały proces wytwarzania oprogramowania.
4. **Lepsza komunikacja w zespole:**
    - Wczesna współpraca między testerami, programistami i interesariuszami pozwala na lepsze zrozumienie wymagań.

Wyzwania w stosowaniu Shift-Left

1. **Zmiana mentalności zespołów:**
    - Zespoły muszą dostosować się do pracy w bardziej zintegrowany sposób i uwzględniać testowanie już na wczesnych etapach.
2. **Potrzeba odpowiednich narzędzi i umiejętności:**
    - Wymaga to automatyzacji testów, narzędzi do zarządzania testami oraz wyszkolonych testerów i programistów.
3. **Zwiększone zaangażowanie w początkowych fazach:**
    - Wczesne angażowanie testerów wymaga więcej zasobów na początku projektu, co może być trudne w tradycyjnych modelach.

Podsumowanie

Podejście **Shift-Left** to strategia testowania, która pozwala na wcześniejsze wykrycie i eliminację defektów, co przekłada się na redukcję kosztów i poprawę jakości oprogramowania. Jest to szczególnie efektywne w środowiskach Agile i DevOps, gdzie iteracyjny rozwój i automatyzacja są kluczowe.

2.1.6. Retrospektywy i doskonalenie procesów

Retrospektywy to regularne spotkania, podczas których zespół analizuje swoją pracę, identyfikuje, co działało dobrze, co mogłoby działać lepiej, oraz opracowuje konkretne działania usprawniające procesy. Są one kluczowym elementem doskonalenia procesów w podejściach zwinnych, ale mogą być stosowane także w innych modelach rozwoju oprogramowania.

Cele retrospektyw

1. **Analiza dotychczasowych działań:**
    - Ustalenie, co funkcjonowało dobrze (tzw. dobre praktyki).
    - Identyfikacja problemów i obszarów wymagających poprawy.
2. **Wspieranie współpracy:**
    - Budowanie zaufania i otwartej komunikacji w zespole.
    - Zachęcanie do dzielenia się pomysłami i refleksjami.
3. **Wprowadzenie usprawnień:**
    - Wypracowanie konkretnych działań, które mogą poprawić efektywność zespołu i jakość pracy.

Kluczowe aspekty retrospektyw

1. **Cykliczność:**
    - Retrospektywy odbywają się regularnie, np. po każdym sprincie w Agile, co pozwala na ciągłe doskonalenie procesu.
2. **Zaangażowanie zespołu:**
    - Wszyscy członkowie zespołu biorą aktywny udział, dzieląc się swoimi spostrzeżeniami i pomysłami.
3. **Skupienie na procesie:**
    - Retrospektywy koncentrują się na procesach pracy, a nie na personalnych błędach.
4. **Kroki retrospektywy:**
    - **Przygotowanie:** Zebranie danych i ustalenie agendy.
    - **Przeprowadzenie retrospektywy:** Analiza działań i identyfikacja wniosków.
    - **Działania następcze:** Ustalenie konkretnych usprawnień do wdrożenia w przyszłości.

Korzyści retrospektyw

1. **Poprawa jakości:**
    - Identyfikowanie i eliminowanie problemów prowadzi do lepszego produktu końcowego.
2. **Zwiększenie efektywności zespołu:**
    - Usprawnienie procesów pracy pozwala zespołowi działać szybciej i bardziej efektywnie.
3. **Lepsza komunikacja:**
    - Retrospektywy sprzyjają otwartej wymianie myśli, co wzmacnia współpracę w zespole.
4. **Budowanie kultury doskonalenia:**
    - Regularne retrospektywy promują podejście „ciągłego uczenia się” i adaptacji do zmieniających się warunków.

Retrospektywy a testowanie

W kontekście testowania retrospektywy:

- **Umożliwiają analizę działań testowych:** Zespół testowy może omówić, które metody, narzędzia i podejścia były skuteczne.
- **Pomagają w identyfikacji problemów:** Na przykład trudności w automatyzacji testów lub problemy z danymi testowymi.
- **Proponują usprawnienia:** Mogą to być lepsze strategie testowe, bardziej efektywne narzędzia lub zmiany w komunikacji między programistami a testerami.

Podsumowanie

Retrospektywy to narzędzie ciągłego doskonalenia, które pozwala zespołom systematycznie analizować swoją pracę i wprowadzać usprawnienia. Dzięki nim procesy testowe i rozwijania oprogramowania stają się bardziej efektywne, a współpraca w zespole – bardziej harmonijna.

2.2.1. Poziomy testów

1. Testy jednostkowe (Unit Testing)

Cel:

- Weryfikacja poprawności działania najmniejszych elementów kodu, takich jak funkcje, metody czy klasy.

Cechy:

- Realizowane przez programistów.
- Skupiają się na szczegółowych testach logicznych i algorytmicznych.
- Często automatyzowane przy użyciu narzędzi takich jak JUnit, NUnit czy pytest.

Korzyści:

- Wczesne wykrywanie błędów w kodzie.
- Zapewnienie, że pojedyncze moduły działają zgodnie z oczekiwaniami.

2. Testy integracyjne (Integration Testing)

Cel:

- Sprawdzenie interakcji pomiędzy różnymi modułami lub komponentami systemu.

Cechy:

- Testowanie komunikacji między jednostkami (np. API, bazy danych, interfejsy).
- Może być realizowane w różnych podejściach, np.:
    - **Big-Bang:** Wszystkie moduły integrowane i testowane jednocześnie.
    - **Incremental:** Testowanie odbywa się stopniowo, np. metoda góra-dół (top-down) lub dół-góra (bottom-up).

Korzyści:

- Zapewnienie, że moduły współdziałają prawidłowo.

- Wykrywanie błędów w integracji, takich jak niezgodności danych lub problemy z komunikacją.

3. Testy systemowe (System Testing)

Cel:

- Sprawdzenie działania całego systemu jako jednego spójnego produktu w odniesieniu do wymagań funkcjonalnych i niefunkcjonalnych.

Cechy:

- Realizowane na kompletnym, zintegrowanym systemie.
- Obejmuje testy funkcjonalne (np. zgodność z wymaganiami) oraz niefunkcjonalne (np. wydajność, użyteczność).
- Wykonywane przez testerów niezależnych od programistów.

Korzyści:

- Weryfikacja, czy system działa zgodnie z założeniami biznesowymi i technicznymi.
- Identyfikacja problemów, które mogą nie być widoczne na wcześniejszych poziomach testów.

4. Testy akceptacyjne (Acceptance Testing)

Cel:

- Ostateczna weryfikacja, czy system spełnia wymagania użytkownika i jest gotowy do wdrożenia.

Cechy:

- Wykonywane przez użytkowników końcowych, klienta lub dedykowany zespół.
- Obejmują różne rodzaje testów akceptacyjnych, np.:
    - **Biznesowe (Business Acceptance Testing):** Czy system spełnia potrzeby biznesowe.
    - **UAT (User Acceptance Testing):** Testy wykonywane przez przyszłych użytkowników.
    - **Regresja akceptacyjna:** Testy ponownego sprawdzenia po poprawkach.

Korzyści:

- Potwierdzenie, że system jest gotowy do wdrożenia.
- Upewnienie się, że wymagania użytkowników zostały spełnione.

Podsumowanie

Każdy poziom testów ma unikalne cele i charakterystyki, które razem zapewniają kompleksową ocenę jakości oprogramowania:

1. **Jednostkowe:** Weryfikacja najmniejszych elementów.
2. **Integracyjne:** Sprawdzanie interakcji między modułami.
3. **Systemowe:** Testowanie całościowego działania systemu.
4. **Akceptacyjne:** Potwierdzenie gotowości do wdrożenia.

Połączenie tych poziomów testów minimalizuje ryzyko błędów i zwiększa pewność co do jakości produktu.

2.2.2. Typy testów

Typy testów określają różne aspekty oprogramowania, które są sprawdzane w procesie testowania. Można je podzielić na testy funkcjonalne, niefunkcjonalne oraz związane z technikami konserwacyjnymi, takie jak testy regresyjne czy testy związane z migracją danych.

1. Testy funkcjonalne

Cel:

- Sprawdzenie, czy system spełnia wymagania funkcjonalne i realizuje oczekiwaną funkcjonalność.

Cechy:

- Skupiają się na tym, *co system robi*, a nie na jego wydajności czy bezpieczeństwie.
- Testowane są funkcje systemu, w tym przetwarzanie danych wejściowych i wyjściowych.
- Bazują na wymaganiach biznesowych, specyfikacjach lub przypadkach użycia.

Przykłady:

- Testowanie poprawności logowania użytkownika.
- Weryfikacja, czy dane wprowadzone w formularzu są prawidłowo zapisane w bazie danych.

Metody:

- Testy oparte na przypadkach testowych (np. na podstawie specyfikacji wymagań).

2. Testy niefunkcjonalne

Cel:

- Sprawdzenie, jak system działa, w tym jego wydajność, użyteczność, bezpieczeństwo i inne cechy jakościowe.

Typy testów niefunkcjonalnych:

- **Wydajnościowe (Performance Testing):** Sprawdzają, jak system radzi sobie pod obciążeniem (np. liczba równoczesnych użytkowników).
- **Użyteczności (Usability Testing):** Ocena, jak łatwo użytkownicy mogą korzystać z systemu.
- **Bezpieczeństwa (Security Testing):** Weryfikacja ochrony danych i odporności na ataki.
- **Kompatybilności (Compatibility Testing):** Sprawdzenie, czy system działa poprawnie na różnych urządzeniach, przeglądarkach czy systemach operacyjnych.
- **Dostępności (Accessibility Testing):** Ocena dostępności dla użytkowników z różnymi ograniczeniami (np. zgodność z WCAG).

Przykłady:

- Pomiar czasu odpowiedzi aplikacji przy dużym obciążeniu.
- Testowanie, czy aplikacja działa na różnych wersjach przeglądarek internetowych.

1. Testy czarnoskrzynkowe (Black-box testing)

Cel:

- Sprawdzenie działania oprogramowania na podstawie jego wymagań i specyfikacji, bez wiedzy o jego wewnętrznej strukturze czy kodzie źródłowym.

Cechy:

- Tester analizuje tylko to, co widzi „z zewnątrz” – dane wejściowe i oczekiwane wyjścia.
- Oparte na analizie funkcjonalności systemu.
- Używane głównie w testach funkcjonalnych.

Techniki:

- **Partycjonowanie na klasy równoważności:** Dzielenie możliwych danych wejściowych na grupy (klasy), w których wszystkie wartości są traktowane jako równoważne.
- **Analiza wartości brzegowych:** Testowanie zachowania systemu na granicach dopuszczalnych danych wejściowych.
- **Tablice decyzyjne:** Użycie tabel do przedstawienia reguł biznesowych i możliwych wyników w różnych kombinacjach danych wejściowych.
- **Przypadki użycia:** Testowanie na podstawie scenariuszy użytkownika opisujących interakcje z systemem.

Przykłady:

- Sprawdzanie, czy system poprawnie przetwarza dane wprowadzone w formularzu.
- Testowanie działania wyszukiwarki na stronie internetowej.

2. Testy białoskrzynkowe (White-box testing)

Cel:

- Weryfikacja działania systemu z uwzględnieniem jego wewnętrznej struktury, logiki i kodu źródłowego.

Cechy:

- Tester ma dostęp do kodu źródłowego i zna jego strukturę.
- Oparte na analizie logicznej i strukturalnej kodu.
- Często stosowane na poziomie testów jednostkowych i integracyjnych.

Techniki:

- **Pokrycie instrukcji (Statement Coverage):** Testowanie każdej instrukcji w kodzie przynajmniej raz.
- **Pokrycie gałęzi (Branch Coverage):** Weryfikacja, czy każda możliwa ścieżka warunkowa została przetestowana.
- **Pokrycie ścieżek (Path Coverage):** Sprawdzenie wszystkich możliwych ścieżek w kodzie.

Przykłady:

- Testowanie algorytmu sortowania, aby sprawdzić, czy obsługuje wszystkie możliwe przypadki.
- Analiza kodu odpowiedzialnego za autoryzację użytkowników.

Porównanie testów czarnoskrzynkowych i białoskrzynkowych

Cecha	Testy czarnoskrzynkowe	Testy białoskrzynkowe
Podejście	Oparte na wymaganiach i specyfikacjach	Oparte na strukturze kodu
Dostęp do kodu	Nie jest wymagany	Wymagany
Poziom testów	Głównie systemowe i akceptacyjne	Głównie jednostkowe i integracyjne
Cel	Sprawdzanie funkcjonalności	Weryfikacja wewnętrznej logiki i działania

2.2.2. Typy testów – ISTQB FL 4.0

W tym rozdziale opisano dwa główne podejścia do testowania oprogramowania, które różnią się metodologią i dostępem do szczegółów technicznych systemu:

1. Testy czarnoskrzynkowe (Black-box testing)

Cel:

- Sprawdzenie działania oprogramowania na podstawie jego wymagań i specyfikacji, bez wiedzy o jego wewnętrznej strukturze czy kodzie źródłowym.

Cechy:

- Tester analizuje tylko to, co widzi „z zewnątrz” – dane wejściowe i oczekiwane wyjścia.
- Oparte na analizie funkcjonalności systemu.
- Używane głównie w testach funkcjonalnych.

Techniki:

- **Partycjonowanie na klasy równoważności:** Dzielenie możliwych danych wejściowych na grupy (klasy), w których wszystkie wartości są traktowane jako równoważne.
- **Analiza wartości brzegowych:** Testowanie zachowania systemu na granicach dopuszczalnych danych wejściowych.
- **Tablice decyzyjne:** Użycie tabel do przedstawienia reguł biznesowych i możliwych wyników w różnych kombinacjach danych wejściowych.
- **Przypadki użycia:** Testowanie na podstawie scenariuszy użytkownika opisujących interakcje z systemem.

Przykłady:

- Sprawdzanie, czy system poprawnie przetwarza dane wprowadzone w formularzu.
- Testowanie działania wyszukiwarki na stronie internetowej

2. Testy białoskrzynkowe (White-box testing)

Cel:

- Weryfikacja działania systemu z uwzględnieniem jego wewnętrznej struktury, logiki i kodu źródłowego.

Cechy:

- Tester ma dostęp do kodu źródłowego i zna jego strukturę.
- Oparte na analizie logicznej i strukturalnej kodu.

- Często stosowane na poziomie testów jednostkowych i integracyjnych.

Techniki:

- **Pokrycie instrukcji (Statement Coverage):** Testowanie każdej instrukcji w kodzie przynajmniej raz.
- **Pokrycie gałęzi (Branch Coverage):** Weryfikacja, czy każda możliwa ścieżka warunkowa została przetestowana.
- **Pokrycie ścieżek (Path Coverage):** Sprawdzenie wszystkich możliwych ścieżek w kodzie.

Przykłady:

- Testowanie algorytmu sortowania, aby sprawdzić, czy obsługuje wszystkie możliwe przypadki.
- Analiza kodu odpowiedzialnego za autoryzację użytkowników.

Porównanie testów czarnoskrzynkowych i białoskrzynkowych

Cecha	Testy czarnoskrzynkowe	Testy białoskrzynkowe
Podejście	Oparte na wymaganiach i specyfikacjach	Oparte na strukturze kodu
Dostęp do kodu	Nie jest wymagany	Wymagany
Poziom testów	Głównie systemowe i akceptacyjne	Głównie jednostkowe i integracyjne
Cel	Sprawdzanie funkcjonalności	Weryfikacja wewnętrznej logiki i działania

Podsumowanie

- **Testy czarnoskrzynkowe** pozwalają ocenić funkcjonalność systemu z perspektywy użytkownika, bez zagłębiania się w jego wewnętrzne działanie.
- **Testy białoskrzynkowe** umożliwiają analizę wewnętrznej struktury kodu, co pomaga zidentyfikować błędy na poziomie logicznym i technicznym.

Oba typy testów wzajemnie się uzupełniają, zapewniając kompleksową ocenę jakości oprogramowania.

2.2.3. Testowanie potwierdzające i testowanie regresji

W tym rozdziale omówiono dwa kluczowe rodzaje testów, które są używane podczas sprawdzania poprawności poprawek oraz wpływu wprowadzonych zmian w oprogramowaniu.

1. Testowanie potwierdzające (Confirmation Testing)

Cel:

- Weryfikacja, czy wcześniej zidentyfikowany błąd został skutecznie naprawiony.

Cechy:

- Dotyczy wyłącznie poprawionego obszaru kodu.
- Testy są powtarzane z wykorzystaniem tych samych danych, które wcześniej ujawniły błąd.
- Wykonuje się je natychmiast po naniesieniu poprawek.

Przykład:

- Jeśli błąd związany z nieprawidłowym logowaniem został zgłoszony i poprawiony, testowanie potwierdzające sprawdza, czy użytkownik może teraz prawidłowo zalogować się do systemu.

2. Testowanie regresji (Regression Testing)

Cel:

- Weryfikacja, czy zmiany wprowadzone w kodzie (np. poprawki błędów lub dodanie nowych funkcji) nie spowodowały niezamierzonych problemów w innych częściach systemu.

Cechy:

- Obejmuje testowanie istniejącej funkcjonalności, aby upewnić się, że nadal działa poprawnie.
- Wykonywane po każdej modyfikacji kodu, niezależnie od jej zakresu.
- Często automatyzowane, aby skrócić czas testowania (np. przy użyciu narzędzi takich jak Selenium, JUnit).

Przykład:

- Po naprawieniu błędu związanego z logowaniem testy regresji sprawdzają, czy inne funkcje, takie jak rejestracja lub reset hasła, nadal działają prawidłowo.

Porównanie testowania potwierdzającego i regresji

Cecha	Testowanie potwierdzające	Testowanie regresji
Cel	Sprawdzenie skuteczności naprawy błędu	Upewnienie się, że inne funkcjonalności nie zostały uszkodzone
Zakres	Ograniczony do naprawionego błędu	Obejmuje całą funkcjonalność systemu
Czas wykonania	Po poprawieniu konkretnego błędu	Po każdej zmianie w kodzie
Automatyzacja	Rzadko automatyzowane	Często automatyzowane

Zastosowanie w praktyce

- **Testy potwierdzające** są szybkim sprawdzeniem poprawek błędów i zwykle realizowane są manualnie lub przy użyciu prostych narzędzi.
- **Testy regresji** są istotne w większych projektach, gdzie zmiany w jednym obszarze mogą mieć wpływ na inne. Automatyzacja odgrywa tu kluczową rolę, zwłaszcza przy dużej liczbie testów do wykonania.

Oba rodzaje testów są niezbędne, aby zapewnić, że wprowadzone zmiany poprawiają jakość systemu, a jednocześnie nie powodują nowych problemów.

2.3. Testowanie pielęgnacyjne

Testowanie pielęgnacyjne (ang. **Maintenance Testing**) jest kluczowym elementem procesu utrzymania oprogramowania. Dotyczy testowania oprogramowania po jego wdrożeniu, aby

upewnić się, że wprowadzane zmiany lub modyfikacje nie wpływają negatywnie na istniejącą funkcjonalność systemu.

Kiedy wykonywane jest testowanie pielęgnacyjne?

Testowanie pielęgnacyjne przeprowadza się w następujących sytuacjach:

1. **Modyfikacje funkcjonalne:** Wprowadzanie nowych funkcji lub ulepszanie istniejących funkcjonalności.
2. **Korekty:** Naprawa wykrytych błędów lub problemów w działającym oprogramowaniu.
3. **Migracje:** Przenoszenie systemu na nowe platformy, systemy operacyjne lub środowiska.
4. **Wycofywanie funkcji:** Usuwanie przestarzałych lub niepotrzebnych elementów systemu.

Cele testowania pielęgnacyjnego

1. **Weryfikacja zmian:** Upewnienie się, że nowe funkcjonalności zostały poprawnie wdrożone.
2. **Zapobieganie regresjom:** Sprawdzenie, czy wprowadzone zmiany nie spowodowały niezamierzonych problemów w innych obszarach systemu.
3. **Ocena stabilności systemu:** Potwierdzenie, że system działa poprawnie w zmienionym środowisku.

Kluczowe działania w testowaniu pielęgnacyjnym

1. **Analiza wpływu (Impact Analysis):**
    - Przed testowaniem ocenia się, które obszary systemu mogą być potencjalnie narażone na problemy w wyniku zmian.
    - Pomaga to zoptymalizować zakres testowania, skupiając się na najbardziej krytycznych elementach.
2. **Testy regresji:**
    - Podstawowy element testowania pielęgnacyjnego.
    - Automatyzacja testów regresji odgrywa kluczową rolę w utrzymaniu efektywności.
3. **Testy potwierdzające:**
    - Weryfikacja, że błędy zostały naprawione zgodnie z wymaganiami.
4. **Testy zgodności:**
    - Sprawdzanie działania systemu w nowym środowisku (np. po aktualizacji systemu operacyjnego).

Wyzwania w testowaniu pielęgnacyjnym

- **Złożoność:** Systemy często mają skomplikowane zależności, które mogą utrudniać ocenę wpływu zmian.
- **Ograniczenia czasowe:** Testowanie pielęgnacyjne często odbywa się w krótkim czasie, aby minimalizować przestoje w działaniu systemu.
- **Brak dokumentacji:** W starszych systemach może brakować aktualnych dokumentów technicznych, co utrudnia analizę zmian.

Podsumowanie

Testowanie pielęgnacyjne jest niezbędnym procesem w cyklu życia oprogramowania, szczególnie w przypadku długotrwale użytkowanych systemów. Obejmuje ono testy regresji, testy zgodności oraz analizę wpływu zmian. Dzięki niemu organizacje mogą utrzymywać stabilność i jakość swoich systemów, nawet w dynamicznie zmieniającym się środowisku.

3.1. Podstawy testowania statycznego

Testowanie statyczne to technika testowania oprogramowania, która polega na analizie artefaktów (np. kodu, dokumentacji, wymagań) bez faktycznego wykonywania programu. Celem jest wykrycie błędów i problemów już na wczesnym etapie rozwoju, co pozwala zaoszczędzić czas i zasoby.

Kluczowe cechy testowania statycznego

1. **Bez wykonywania kodu:** W przeciwieństwie do testowania dynamicznego, w testowaniu statycznym oprogramowanie nie jest uruchamiane.
2. **Analiza artefaktów:** Testowaniu poddaje się dokumenty, specyfikacje, kod źródłowy i inne elementy procesu wytwarzania oprogramowania.
3. **Efektywność kosztowa:** Wykrycie błędów na etapie testowania statycznego jest znacznie tańsze niż po wdrożeniu systemu.

Metody testowania statycznego

1. Przeglądy (Reviews):

- Proces polegający na ręcznym przeglądaniu artefaktów w celu znalezienia błędów, niespójności lub braków.
- Typy przeglądów:
    - **Nieformalne przeglądy:** Niezorganizowane, często ad-hoc, bez formalnych procedur.
    - **Przeglądy grupowe (np. Fagan Inspection):** Formalny proces z wyznaczonymi rolami, takimi jak autor, recenzent i moderator.

2. Analiza statyczna:

- Automatyczne narzędzia analizują kod źródłowy w celu wykrycia potencjalnych problemów, takich jak:
    - Naruszenia standardów kodowania.
    - Nieoptymalne konstrukcje.
    - Potencjalne błędy (np. zmienne niezainicjowane).
- Przykładowe narzędzia: SonarQube, Checkstyle.

Korzyści testowania statycznego

- **Wczesne wykrywanie problemów:** Błędy są identyfikowane przed fazą testowania dynamicznego, co zmniejsza ryzyko kosztownych poprawek na późniejszych etapach.
- **Poprawa jakości dokumentacji:** Przeglądy pomagają znaleźć nieścisłości i brakujące informacje w wymaganiach i specyfikacjach.
- **Zapewnienie zgodności z wymaganiami:** Analiza pomaga upewnić się, że system spełnia określone standardy i normy.

- **Skalowalność:** Narzędzia automatyzujące analizę statyczną mogą szybko analizować duże zbiory kodu.

Przykład zastosowania testowania statycznego

1. **Przegląd wymagań:** Zespół projektowy analizuje dokumentację wymagań, aby upewnić się, że jest kompletna, zrozumiała i pozbawiona sprzeczności.
2. **Analiza kodu:** Narzędzie automatyczne wykrywa błędy w kodzie, takie jak nieużywane zmienne lub potencjalne problemy z bezpieczeństwem.

Podsumowanie

Testowanie statyczne pozwala na szybkie i efektywne wykrycie błędów w dokumentacji, kodzie czy specyfikacjach, zanim system zostanie uruchomiony. Dzięki temu zwiększa się jakość oprogramowania i redukuje ryzyko kosztownych błędów na późniejszych etapach cyklu życia oprogramowania.

3.1.3. Różnice między testowaniem statycznym a dynamicznym

Testowanie statyczne i dynamiczne to dwie różne metody zapewnienia jakości oprogramowania, które wzajemnie się uzupełniają. W rozdziale 3.1.3 sylabus ISTQB FL wyjaśnia kluczowe różnice między nimi, zarówno pod względem podejścia, jak i celu.

1. Definicja i podejście

- **Testowanie statyczne:**
    - Analiza artefaktów (np. kodu, dokumentacji, wymagań) bez uruchamiania programu.
    - Wykorzystywane narzędzia: przeglądy, inspekcje, analiza kodu za pomocą narzędzi automatycznych.
- **Testowanie dynamiczne:**
    - Wymaga uruchomienia programu w celu sprawdzenia jego zachowania w rzeczywistych warunkach.
    - Wykorzystywane narzędzia: frameworki testowe, środowiska symulacyjne.

2. Cele

- **Testowanie statyczne:**
    - Wykrywanie błędów w dokumentach, kodzie i specyfikacjach na wczesnym etapie.
    - Zapewnienie zgodności z normami i standardami (np. stylu kodowania).
- **Testowanie dynamiczne:**
    - Weryfikacja działania oprogramowania w praktyce, w tym jego funkcjonalności, wydajności i użyteczności.
    - Sprawdzanie, czy system spełnia wymagania użytkownika.

3. Przykłady zastosowania

- **Testowanie statyczne:**
    - Przegląd wymagań, aby znaleźć niejasności lub sprzeczności.

        - Analiza kodu w celu wykrycia błędów, takich jak zmienne niezainicjowane lub naruszenia zasad bezpieczeństwa.
- **Testowanie dynamiczne:**
    - Uruchamianie przypadków testowych, aby sprawdzić, czy funkcja zwraca oczekiwany wynik.
    - Testy wydajnościowe, aby ocenić, czy system radzi sobie pod dużym obciążeniem.

4. Narzędzia

- **Statyczne:**
    - Narzędzia do analizy kodu, np. SonarQube, Checkstyle.
    - Narzędzia wspierające przeglądy, np. Collaborative Review Tools.
- **Dynamiczne:**
    - Frameworki testowe, np. Selenium, JUnit.
    - Narzędzia do monitorowania wydajności, np. JMeter.

5. Zalety i ograniczenia

Cecha	Testowanie statyczne	Testowanie dynamiczne
Etap wykrywania błędów	Na wczesnych etapach (przed uruchomieniem kodu).	Po wdrożeniu funkcjonalności.
Koszt wykrycia błędu	Niższy (wcześnie wykryte błędy są tańsze w naprawie).	Wyższy (błędy wykryte późno są kosztowne).
Rodzaj błędów	Logiczne, stylistyczne, niezgodności z wymaganiami.	Błędy funkcjonalne, wydajnościowe.
Skuteczność	Nie wykrywa problemów w działaniu systemu.	Nie wykrywa problemów w specyfikacjach czy kodzie bez uruchomienia.

Podsumowanie

- **Testowanie statyczne** pozwala na wczesne wykrywanie błędów w dokumentach i kodzie, co obniża koszty ich naprawy.
- **Testowanie dynamiczne** sprawdza, jak system działa w praktyce, identyfikując błędy funkcjonalne i wydajnościowe.
- Oba podejścia są niezbędne, ponieważ razem pokrywają różne aspekty jakości oprogramowania.

3.2.1. Korzyści wynikające z wczesnego i częstego otrzymywania informacji zwrotnych od interesariuszy

Wczesne i częste informacje zwrotne od interesariuszy pozwalają szybko zauważyć i naprawić błędy w wymaganiach czy projektach, zanim staną się kosztowne. Dzięki temu zespół lepiej rozumie, czego oczekują użytkownicy, i może szybciej dostosować produkt do ich potrzeb. To pomaga uniknąć problemów później, poprawia współpracę z klientami i ułatwia wprowadzanie zmian w trakcie projektu.

3.2.2. Czynności wykonywane w procesie przeglądu

Proces przeglądu w testowaniu statycznym składa się z kilku kluczowych czynności, które pomagają w wykrywaniu błędów i doskonaleniu jakości artefaktów (np. dokumentów, kodu). Oto główne etapy:

1. **Planowanie przeglądu**
    - Określenie celu, zakresu i harmonogramu przeglądu, a także wyznaczenie osób odpowiedzialnych za przeprowadzenie przeglądu.
2. **Przygotowanie**
    - Recenzenci zapoznają się z materiałami przed przeglądem, aby mogli skutecznie zidentyfikować problemy.
3. **Spotkanie przeglądowe**
    - Zespół przeglądowy spotyka się, aby omówić znalezione błędy. Spotkanie to może być formalne lub mniej sformalizowane w zależności od rodzaju przeglądu.
4. **Raportowanie wyników**
    - Zapisanie znalezionych problemów i zaproponowanie poprawek. Może to obejmować sporządzenie raportu z przeglądu.
5. **Działania następcze**
    - Wdrożenie poprawek i śledzenie działań naprawczych w celu poprawy jakości produktu.

Każda z tych czynności ma na celu wykrycie błędów, poprawę jakości oraz wczesne identyfikowanie potencjalnych problemów, zanim staną się one kosztowne.

3.2.3. Role i obowiązki w przeglądach

1. **Autor**
    - Odpowiada za stworzenie artefaktu (np. dokumentu, kodu), który jest przeglądany.
    - Przygotowuje materiał i wprowadza poprawki na podstawie uwag z przeglądu.
2. **Kierownik przeglądu**
    - Odpowiada za organizację i zarządzanie procesem przeglądu.
    - Zapewnia odpowiednią komunikację między wszystkimi uczestnikami i czuwa nad terminowością oraz efektywnością przeglądu.
3. **Moderator**
    - Koordynuje spotkanie przeglądowe.
    - Utrzymuje porządek, zarządza czasem i zapewnia, aby przegląd przebiegał zgodnie z planem i był konstruktywny.
4. **Protokolant**
    - Zapisuje wyniki przeglądu, w tym wykryte błędy oraz podjęte decyzje.
    - Sporządza raport z przeglądu, dokumentując działania następcze.
5. **Przeglądający (Recenzenci)**
    - Osoby, które analizują artefakt pod kątem błędów i sugerują ulepszenia.

- Wnoszą swoją wiedzę, identyfikują problemy i pomagają w poprawie jakości materiału.

6. **Lider przeglądu**
    - Może pełnić rolę główną w bardziej formalnych przeglądach, takich jak inspekcje.
    - Odpowiada za całokształt procesu przeglądu, zarządza zespołem i podejmuje decyzje dotyczące wyników przeglądu.

Każda z tych ról pełni kluczową funkcję w przeglądzie, zapewniając, że proces jest dobrze zorganizowany, efektywny i prowadzi do wykrywania oraz usuwania błędów na wczesnym etapie.

3.2.4. Typy przeglądów

Przegląd nieformalny

- Jest to najprostsza forma przeglądu, odbywająca się bez formalnych reguł i przygotowań.
- Celem jest szybkie zidentyfikowanie problemów w artefaktach w nieformalnej atmosferze, bez potrzeby szczegółowej dokumentacji.

2. **Przejrzenie (Walkthrough)**
    - Jest to proces, w którym autor artefaktu prowadzi zespół przez dokument lub kod, omawiając jego zawartość.
    - Uczestnicy przeglądu zadają pytania i dzielą się swoimi uwagami w celu identyfikacji błędów, nieścisłości czy innych problemów.
    - Jest to mniej formalne niż inspekcja, ale bardziej strukturalne niż przegląd nieformalny.
3. **Przegląd techniczny**
    - Ma charakter bardziej formalny niż przejrzenie, z udziałem ekspertów technicznych, którzy analizują dokumenty w celu wykrycia błędów technicznych, takich jak problemy z architekturą, kodem czy implementacją.
    - Może obejmować różne aspekty techniczne, ale nie jest tak szczegółowy jak inspekcja.
4. **Inspekcja**
    - Jest to najbardziej formalna forma przeglądu.
    - Przegląd odbywa się według ścisłego planu, obejmującego przygotowanie, spotkanie, raportowanie wyników i działania naprawcze.
    - Zespół inspekcyjny składa się z różnych ról (autor, recenzenci, moderator, protokolant), a wyniki są dokładnie dokumentowane.

Każdy z tych typów przeglądów różni się stopniem formalności, przygotowania oraz szczegółowości analizy artefaktów.

4.1. Ogólna charakterystyka technik testowania

Techniki testowania pomagają testerom w przeprowadzaniu analizy testów (która odpowiada na pytanie „co należy przetestować”) oraz w projektowaniu testów (które odpowiada na pytanie „jak należy testować”).

1. Techniki oparte na specyfikacji (czarnoskrzynkowe)

- Skupiają się na **zewnętrznym zachowaniu systemu**, bez znajomości jego wewnętrznej budowy.
- Przykłady: testowanie wartości brzegowych, klas równoważności, tablic decyzyjnych.
- Stosowane głównie na wyższych poziomach testów (np. testach systemowych, akceptacyjnych).

2. Techniki oparte na strukturze (białoskrzynkowe)

- Wymagają znajomości **wewnętrznej logiki** kodu lub komponentu.
- Przykład: pokrycie instrukcji (czy każda linia kodu została wykonana), pokrycie decyzji.
- Używane głównie przez programistów przy testach jednostkowych.

3. Techniki oparte na doświadczeniu

- Opierają się na **intuicji, wiedzy i doświadczeniu** testera.
- Przykłady: testowanie eksploracyjne, atakowanie typowych błędów.
- Są przydatne, gdy dokumentacja jest niepełna lub czas na przygotowanie testów jest ograniczony.

Kluczowy przekaz:

Każda z technik ma swoje zastosowania i warto je **łączyć**, by zwiększyć skuteczność testowania – np. używać technik opartych na specyfikacji do pokrycia wymagań, struktur do sprawdzania kodu, a doświadczenia do wychwycenia nietypowych błędów.

4.2. Czarnoskrzynkowe techniki testowania – ISTQB FL 4.0

Czarnoskrzynkowe techniki testowania (ang. *black-box testing*) polegają na sprawdzaniu działania systemu **z zewnątrz**, bez znajomości jego wewnętrznej struktury (np. kodu źródłowego). Tester skupia się na tym, **co system powinien robić**, a nie *jak* to robi.

Główne techniki czarnoskrzynkowe opisane w sylabusie:

4.2.1. Podział na klasy równoważności (ang. Equivalence Partitioning)

- Dane wejściowe dzieli się na grupy (klasy), które system powinien traktować tak samo.
- Z każdej klasy wybiera się po jednej reprezentatywnej wartości do testów.

Przykład: Jeśli system akceptuje wiek od 18 do 65 lat, tworzymy 3 klasy:

- wiek < 18 (nieważna),
- 18–65 (ważna),
- 65 (nieważna)

4.2.2. Analiza wartości brzegowych (ang. Boundary Value Analysis)

- **Analiza dwupunktowa (two-point boundary analysis)**
  - **Na każdy zakres testujemy tylko wartości graniczne** – czyli **dokładnie na granicy** dopuszczalnych danych wejściowych (np. minimalna i maksymalna wartość).
  - **Typowe wartości testowe:**
    Dla zakresu 18–65: 18 i 65.

Zastosowanie:

- Szybka, uproszczona wersja analizy brzegów.
- Stosowana, gdy testy muszą być szybkie lub zakres danych jest dobrze przetestowany innymi metodami.

- **Analiza trójpunktowa (three-point boundary analysis)**
  - Dla każdej granicy bierze się:
    - wartość **tuż poniżej** granicy (np. 17),
    - wartość **dokładnie na granicy** (np. 18),
    - wartość **tuż powyżej** granicy (np. 19).

Typowe wartości testowe (dla zakresu 18–65):

- Dół: 17, 18, 19
- Góra: 64, 65, 66

Zastosowanie:

- Pełniejsza analiza, **bardziej dokładna i skuteczna w wykrywaniu błędów**, które pojawiają się tuż przy granicach.
- To właśnie ten wariant jest promowany przez ISTQB jako **pełna analiza wartości brzegowych**.

Obie wersje są poprawne i stosowane – wybór zależy od tego, **ile czasu masz na testy** i **jak ważna jest dana funkcja**.

4.2.3. Testowanie w oparciu o tablicę decyzyjną (ang. Decision Tables)

Technika, która pomaga uporządkować złożone reguły decyzyjne. Sprawdza się świetnie, gdy wynik działania systemu zależy od wielu warunków jednocześnie.

Po co się tego używa?

Gdy mamy wiele warunków typu "jeśli… to…" i różne kombinacje tych warunków dają różne wyniki – łatwo coś przeoczyć. Tablica decyzyjna pozwala jasno pokazać **wszystkie możliwe kombinacje** i odpowiednie akcje.

Akcje (Actions):

To, co system ma zrobić – np.:

- Przyznaj 10% rabatu
- Nie przyznawaj rabatu

Przykład tablicy decyzyjnej:

Warunek	Reguła 1	Reguła 2	Reguła 3	Reguła 4
Ma kartę lojalnościową	Prawda	Prawda	Fałsz	Fałsz
Wartość zakupów > 200 zł	Prawda	Fałsz	Prawda	Fałsz
Akcja: Przyznaj 10% rabatu	☑	✗	✗	✗

[!] Widać, że tylko przy spełnieniu obu warunków klient dostaje rabat (Reguła 1).

Zalety:

- Pokazuje wszystkie istotne kombinacje warunków.
- Pomaga wykrywać **braki w wymaganiach** lub **sprzeczności**.
- Idealna do testowania systemów z **złożoną logiką biznesową** (np. formularze, reguły rabatowe, logika workflow).

W testach:

Każda kolumna w tablicy to **jeden przypadek testowy**. Tworzy się ich dokładnie tyle, ile jest **unikalnych reguł decyzyjnych**.

4.3. Białoskrzynkowe techniki testowania

Z uwagi na ich popularność i prostotę w niniejszym podrozdziale skupiono się na dwóch białoskrzynkowych technikach testowania związanych z kodem. Są to:

- testowanie instrukcji;
- testowanie gałęzi.

4.3.1. Testowanie instrukcji i pokrycie instrukcji kodu

- Cel: sprawdzić, czy każda pojedyncza instrukcja w kodzie została wykonana przynajmniej raz.
- Pokrycie nazywa się: pokrycie instrukcji (statement coverage).

◇ **Dlaczego to ważne?**

Jeśli jakaś linijka kodu nigdy się nie wykonuje, może tam być **ukryty błąd**, którego testy nie wykryją.

◇ **Przykład:**

```
x = 10
if x > 0:
    print("Liczba dodatnia")  # ← To chcemy przetestować
```

Aby test pokrył tę instrukcję, `x > 0` musi być prawdą.

4.3.2. Testowanie gałęzi i pokrycie gałęzi

- **Cel:** sprawdzić, czy **każda gałąź w kodzie została przetestowana** – czyli zarówno ścieżka „tak”, jak i „nie” dla każdego warunku (if, else, switch, itd.).
- Pokrycie to: **pokrycie decyzji (decision/branch coverage)**.

◇ **Dlaczego to ważne?**

Bo kod może robić różne rzeczy w zależności od warunku. Trzeba sprawdzić **oba warianty**.

◇ **Przykład:**

```
if x > 0:
    print("Dodatnia")
else:
    print("Niedodatnia")
```

Aby uzyskać pełne pokrycie **gałęzi**, musisz:

- raz dać x > 0 (żeby wejść w if)
- raz dać x <= 0 (żeby wejść w else)

Różnice między nimi:

Cecha	Testowanie instrukcji	Testowanie gałęzi
Co sprawdza	Czy każda linijka była wykonana	Czy każdy warunek był prawdziwy i fałszywy
Ile testów wystarczy?	Mniej	Więcej (bardziej dokładne)
Obejmuje wszystkie ścieżki?	Nie	Częściej tak
Przykład	Tylko if, nie else	if i else

Podsumowanie:

- **Testowanie instrukcji**: Czy każda linijka kodu „zadziałała”?
- **Testowanie gałęzi**: Czy każda możliwa decyzja w kodzie (tak/nie) została sprawdzona?

Oba podejścia pomagają znaleźć błędy w logice kodu – tylko w różnym stopniu dokładności.

4.3.3. Korzyści wynikające z testowania białoskrzynkowego

Testowanie białoskrzynkowe daje kilka **konkretnych korzyści**, zwłaszcza gdy chodzi o jakość kodu i skuteczne znajdowanie błędów „od środka”.

Oto najważniejsze zalety:

1. Dokładność – sprawdzasz cały kod

- Można **upewnić się, że wszystkie instrukcje i decyzje w kodzie zostały wykonane**.
- Dzięki temu nie ma „martwego kodu” – czyli takiego, który nigdy się nie uruchamia.

2. Szybkie wykrywanie błędów w logice

- Umożliwia wykrycie błędów w **warunkach, pętlach i przepływach sterowania**.

- Pomaga np. zauważyć, że jakaś gałąź w kodzie zawsze jest pomijana, mimo że nie powinna.

3. Dobre uzupełnienie testów funkcjonalnych (czarnoskrzynkowych)

- Testy czarnoskrzynkowe badają, **co system robi na zewnątrz**, a białoskrzynkowe **jak to robi wewnątrz**.
- Razem dają **pełniejszy obraz jakości oprogramowania**.

4. Lepsze pokrycie kodu (coverage)

- Dzięki białoskrzynkowemu testowaniu można mierzyć, **ile procent kodu zostało przetestowane**.
- To pozwala ocenić, czy testy są wystarczająco dokładne.

5. Pomoc w optymalizacji i refaktoryzacji

- Ułatwia **znalezienie zbędnego lub zbyt skomplikowanego kodu**, który można uprościć.
- Pomaga też w sprawdzeniu, czy zmiany nie wpłynęły negatywnie na inne części systemu.

Podsumowując:

Testowanie białoskrzynkowe daje wgląd w to, jak działa kod „od środka", pozwala sprawdzić każdą ścieżkę działania programu i wykryć błędy, których testy z zewnątrz mogą nie zauważyć.

4.4. Techniki testowania oparte na doświadczeniu

W testowaniu opartym na doświadczeniu nie korzysta się z dokumentacji ani formalnych przypadków testowych. Tester używa **własnej wiedzy, intuicji i wcześniejszych doświadczeń**, żeby **szukać błędów tam, gdzie mogą się ukrywać**.

To podejście jest szczególnie przydatne, gdy:

- dokumentacja jest niekompletna lub jej brak,
- czas na testowanie jest bardzo ograniczony,
- system jest już znany i warto szukać nietypowych problemów.

4.4.1. Zgadywanie błędów

Zgadywanie błędów to technika testowania oparta na **doświadczeniu i intuicji testera**. Polega na tym, że tester **świadomie zgaduje, gdzie mogą wystąpić błędy** — bazując na:

- własnej wiedzy o podobnych systemach,
- typowych problemach technicznych,
- wcześniejszych usterkach,
- znajomości słabych punktów danego typu aplikacji.

◇ **Przykład:**

Tester testuje formularz logowania. Zamiast tylko wpisywać poprawne dane, sprawdza też:

- co się stanie po wpisaniu pustego hasła,
- co się stanie, jeśli login zawiera niedozwolone znaki,
- czy system pokazuje komunikaty o błędach,
- czy nie da się zalogować po 3 błędnych próbach (np. blokada konta).

Nie potrzebuje do tego szczegółowych przypadków testowych – po prostu **wie, co może pójść nie tak**.

◇ **Zalety:**
- szybkie wykrywanie typowych, często popełnianych błędów,
- nie wymaga dokumentacji,
- elastyczne i można je zastosować niemal od razu.

◇ **Wady:**
- zależy od doświadczenia testera,
- trudne do powtórzenia i udokumentowania,
- nie daje gwarancji pokrycia całego systemu.

Podsumowanie:

Zgadywanie błędów to podejście, w którym tester „celuje” w miejsca potencjalnie problematyczne – bazując na wcześniejszych projektach, intuicji i znajomości systemów.

4.4.2. Testowanie eksploracyjne

Technika, w której **tester samodzielnie bada system, tworzy i wykonuje testy „na bieżąco”**, bez wcześniejszych, szczegółowych planów testowych. Działa intuicyjnie i elastycznie.

◇ **Kluczowe cechy:**
- **Uczenie się** systemu podczas testowania,
- **Tworzenie testów na bieżąco**, zależnie od tego, co się dzieje w aplikacji,
- **Wykonywanie testów od razu**, bez czekania na gotowe przypadki testowe,
- Tester sam decyduje, **co, jak i kiedy sprawdzić**.

Testowanie eksploracyjne w sesjach a chartery

W testowaniu eksploracyjnym istnieje podejście zwane **Session-Based Exploratory Testing (SBET)**, czyli **testowanie eksploracyjne oparte na sesjach**.

W tym podejściu każda sesja testowa:
- ma **określony cel** (czyli właśnie **charter**),
- trwa zazwyczaj określony czas (np. 60–90 minut),
- kończy się krótkim **raportem z sesji**.

◇ **Czym jest charter w tym kontekście?**

To **krótki opis tego, co tester ma sprawdzić** podczas konkretnej sesji eksploracyjnej. Czyli:

Charter = temat + cel testowania w danej sesji.

Przykład:

„Przetestuj działanie filtrowania produktów po cenie i kategorii na stronie sklepu.”

◇ **Co zawiera typowa sesja eksploracyjna?**

1. **Charter** – co testujemy.
2. **Czas trwania** – np. 90 minut.
3. **Notatki testera** – co działało, co nie, co warto sprawdzić dalej.
4. **Raport końcowy** – np. lista znalezionych defektów, obszary ryzyka.

Podsumowanie:

- **Chartery** to integralna część testowania eksploracyjnego prowadzonego w sesjach.
- Pomagają testerowi **skupić się na konkretnym celu**, ale zostawiają mu dużą swobodę w sposobie działania.
- W sylabusie ISTQB FL wspomniano o nich krótko, ale są **ważnym narzędziem w praktyce testowania eksploracyjnego**.

4.4.3. Testowanie w oparciu o listę kontrolną

Testowanie w oparciu o listę kontrolną (ang. checklist-based testing) polega na tym, że tester korzysta z **gotowej listy punktów do sprawdzenia**, żeby upewnić się, że nic ważnego nie zostało pominięte.

To coś jak lista „do odhaczenia” (checklist), którą tester przechodzi krok po kroku.

◇ **Co znajduje się na liście?**

Lista kontrolna może zawierać:

- elementy interfejsu (np. „czy wszystkie pola formularza są podpisane?”),
- kwestie funkcjonalne (np. „czy możliwe jest anulowanie operacji?”),
- problemy z bezpieczeństwem, wydajnością itd.

◇ **Przykład checklisty:**

Dla strony rejestracji:

- Sprawdź, czy pola mają walidację.
- Sprawdź, czy przycisk „Zarejestruj” działa.
- Sprawdź, czy hasło jest ukrywane (maskowane).
- Sprawdź, czy po błędzie pojawia się czytelny komunikat.

- **Zastosowanie:**
  - W testach eksploracyjnych jako pomocnik.
  - Do testowania regresji (sprawdzanie znanych elementów).
  - Dla osób początkujących – żeby nic nie pominąć.

Podsumowanie:

Testowanie z listą kontrolną to sposób na uporządkowane sprawdzenie systemu — bez potrzeby pisania pełnych przypadków testowych. Dobrze się sprawdza w codziennej pracy i jako uzupełnienie innych technik.

4.5. Podejścia do testowania oparte na współpracy

4.5.1. Wspólne pisanie historyjek użytkownika

Cały zespół – biznes (Product Owner), analityk, programista i tester – **razem** formułuje historyjkę użytkownika, zanim ruszy kodowanie.

Dlaczego warto?

1. **Jedno, wspólne zrozumienie potrzeby**
   • biznes mówi, *po co* to robimy,
   • programista – *czy i jak* to zbudować,
   • tester – *jak sprawdzić, że działa*.
2. **Mniej luk i niejasności**
   • błędy w wymaganiach wychodzą od razu przy rozmowie, a nie po fakcie.
3. **Testowalność od pierwszej chwili**
   • tester dopisuje kryteria akceptacji / scenariusze BDD (Given-When-Then),
   • programista może od razu przygotować testy jednostkowe/TDD.

Jak to wygląda w praktyce?

Krok	Co robimy?	Przykład (aplikacja sklepu)
1. Zbieramy się w trójkę (tzw. *Three Amigos*)	PO, programista i tester omawiają funkcjonalność	Zespół spotyka się, by omówić dodanie produktu do koszyka
2. PO opisuje potrzebę	Formułujemy historyjkę w stylu: *Jako użytkownik...*	„Jako klient chcę dodać produkt do koszyka, żeby móc go później kupić”
3. Uzupełniamy kryteria akceptacji	Wspólnie ustalamy, po czym poznamy, że funkcja działa poprawnie	• Produkt trafia do koszyka• Widać cenę i sumę• Koszyk aktualizuje się poprawnie
4. Tester zgłasza przypadki brzegowe	Pytania typu „a co jeśli…?”	• Co jeśli produkt jest niedostępny?• Co jeśli klient nie jest zalogowany?
5. Programista ocenia wykonalność	Omawiamy techniczne aspekty i zależności	„Potrzebujemy endpointu /POST cart oraz walidacji stanu magazynowego”
6. Finalizujemy historyjkę	Notujemy gotową historyjkę z kryteriami akceptacji	Zapisujemy ją do backlogu w narzędziu (np. Jira, Azure DevOps)

Efekt

- Historyjka spełnia zasadę **INVEST** (Independent, Negotiable, Valuable, Estimable, Small, Testable).
- Zespół startuje z **jednym, wspólnym punktem odniesienia** – bez zgadywania w późniejszej fazie.

Podsumowanie:

wspólne pisanie historyjek to szybka inwestycja w jasne wymagania + gotowe kryteria testowe, dzięki którym unikamy nieporozumień i przyspieszamy cały projekt.

4.5.2. Kryteria akceptacji

To warunki, które muszą być spełnione, aby uznać, że historyjka użytkownika została poprawnie zrealizowana.

◇ **Dlaczego są ważne?**

- Pomagają **zrozumieć wymaganie** (dla PO, devów, testerów),
- Służą jako **baza do pisania testów** (również automatycznych),
- Dzięki nim wiadomo, **co znaczy "skończone"**.

◇ **Kto je ustala?**

Najczęściej wspólnie:
PO + tester + programista → najlepiej podczas pisania historyjek użytkownika.

◇ **Przykład historyjki z kryteriami**

Historyjka:

> Jako klient chcę móc dodać produkt do koszyka, aby go później kupić.

Kryteria akceptacji:

1. Po kliknięciu "Dodaj do koszyka", produkt pojawia się w koszyku.
2. W koszyku widoczna jest nazwa, ilość i cena produktu.
3. Jeśli produkt już jest w koszyku, jego ilość zwiększa się o 1.
4. Jeśli użytkownik nie jest zalogowany, i tak może dodać produkt.

◇ **Format BDD (Given – When – Then)**

Można też zapisać je jako scenariusze testowe:

> **Given** klient przegląda stronę produktu
> **When** kliknie "Dodaj do koszyka"
> **Then** produkt powinien pojawić się w koszyku z poprawną ceną

Podsumowanie:

Kryteria akceptacji to konkretne punkty do odhaczenia, by mieć pewność, że funkcja działa zgodnie z oczekiwaniem użytkownika.

4.5.3. Wytwarzanie sterowane testami akceptacyjnymi (ATDD)

ATDD to podejście, w którym **zanim powstanie kod, cały zespół najpierw tworzy testy akceptacyjne** (czyli kryteria, które muszą być spełnione, by uznać, że funkcja działa poprawnie).

- **Kto bierze w tym udział?**
  - **Product Owner (PO)** – mówi, co ma być zbudowane i czego oczekuje użytkownik,
  - **Tester** – podpowiada, jak to przetestować i wykrywa przypadki brzegowe,
  - **Programista** – mówi, co da się zaimplementować i jak to zrobić.

Wszyscy wspólnie ustalają **co znaczy „działa”** – jeszcze przed pisaniem kodu.

- **Przykład – jak wygląda ATDD w praktyce?**
  1. **Ustalamy wymaganie**:

     > Jako użytkownik chcę dodać produkt do koszyka.

  2. **Tworzymy test akceptacyjny** (np. w formacie BDD – Given–When–Then):

     ```
     Given użytkownik jest na stronie produktu,
     When kliknie "Dodaj do koszyka",
     Then produkt powinien pojawić się w koszyku.
     ```

  3. **Testy są zapisywane (często automatycznie)** zanim powstanie funkcja.
  4. **Dopiero potem programista pisze kod**, tak żeby test przeszedł.
  5. **Testy są uruchamiane automatycznie** jako potwierdzenie, że funkcjonalność działa.

- **Co daje ATDD?**
  - Funkcje są lepiej **przemyślane od początku** – mniej poprawek później,
  - Zespół ma **wspólne zrozumienie celu**,
  - Testy akceptacyjne mogą być **zautomatyzowane** (np. w narzędziach typu Cucumber, Behave).

Podsumowanie:

ATDD = zanim napiszesz kod, ustal, jak udowodnisz, że działa.
To nie tylko testowanie – to sposób współpracy, który zmniejsza błędy i nieporozumienia w zespole.

[!] *Czym są **historyjki użytkownika** (ang. user stories)?*

*To **krótkie, zwięzłe opisy tego, czego chce użytkownik i dlaczego**, pisane prostym, zrozumiałym językiem. Używa się ich głównie w metodykach zwinnych (Agile), np. Scrumie.*

- ***Szablon historyjki użytkownika:***

  Jako *[typ użytkownika]*
  chcę *[coś zrobić]*
  aby *[osiągnąć cel].*

- ***Przykład:***

  Jako *zalogowany klient*
  chcę *dodać produkt do koszyka*
  aby *móc go później kupić.*

- ***Co daje historyjka?***
  - ***Opisuje wartość biznesową****, a nie rozwiązanie techniczne.*
  - *Jest* ***punktem wyjścia do rozmowy*** *z zespołem.*
  - *Służy do* ***planowania pracy*** *i ustalania* ***kryteriów akceptacji****.*
  - *Pomaga tworzyć testy akceptacyjne (np. w ATDD lub BDD).*

- ***Historyjka ≠ szczegółowa specyfikacja***

Historyjki są celowo ***niepełne****, bo mają być omawiane i doprecyzowane wspólnie z zespołem (np. z PO, testerem i programistą).*

Podsumowanie:

Historyjka użytkownika to ***mała historia o potrzebie użytkownika****, którą zespół realizuje krok po kroku, dodając szczegóły w trakcie rozmów.*

5.1. Planowanie testów