Rodzaje cykli życia produktu:

1. Kaskadowy (cykl wodospadu/liniowy)

W modelu kaskadowym proces jest jak wodospad – każda faza (np. analiza, projektowanie, kodowanie, testowanie) kończy się, zanim zacznie się kolejna. Wszystko przebiega w jednym kierunku.

- **Jak się testuje?**

Testowanie odbywa się dopiero po zakończeniu kodowania. To oznacza, że jeśli po drodze pojawi się błąd w wymaganiach lub projekcie, jego wykrycie może nastąpić bardzo późno, co bywa kosztowne i czasochłonne.

- **Plusy:** Prosty i dobrze zorganizowany.
- **Minusy:** Brak elastyczności – trudno wprowadzać zmiany w późnych etapach.

2. Prototypowanie

Tutaj tworzy się szybki prototyp, czyli uproszczoną wersję systemu, aby zobaczyć, czy działa zgodnie z oczekiwaniami użytkownika. Dopiero po zatwierdzeniu prototypu przechodzi się do pełnej implementacji.

- **Jak się testuje?**

Prototyp jest testowany głównie pod kątem funkcjonalności i zgodności z wymaganiami użytkownika. Użytkownicy mogą zgłaszać poprawki na bieżąco.

- **Plusy:** Pozwala szybko zweryfikować pomysły i uniknąć kosztownych błędów.
- **Minusy:** Prototyp może wprowadzać w błąd, sugerując, że produkt końcowy powstanie równie szybko.

3. Programowanie odkrywcze

Polega na tworzeniu oprogramowania bez szczegółowego planu. Programista po prostu zaczyna kodować, bazując na ogólnych założeniach, a szczegóły pojawiają się w trakcie pracy.

- **Jak się testuje?**

Testowanie odbywa się równolegle z programowaniem, na bieżąco. Jest bardziej chaotyczne, bo zmiany mogą pojawiać się niespodziewanie.

- **Plusy:** Elastyczne i szybkie.
- **Minusy:** Może prowadzić do niedopracowanego produktu, bo brakuje planu i struktury.

4. Montaż z gotowych elementów

Tutaj buduje się system, używając istniejących komponentów (np. bibliotek, modułów, narzędzi). Tworzy się coś w rodzaju „składaka”.

- **Jak się testuje?**

Najważniejsze jest sprawdzanie integracji, czyli tego, czy elementy działają razem bez konfliktów. Testuje się też, czy gotowe elementy spełniają wymagania projektu.

- **Plusy:** Szybsze tworzenie systemu, bo wykorzystuje się już gotowe elementy.
- **Minusy:** Może być problematyczne, jeśli komponenty nie współpracują dobrze lub mają ograniczenia.

5. Cykl spiralny

To hybryda prototypowania i modelu kaskadowego. Praca odbywa się w iteracjach (spiralach), gdzie każda pętla dodaje nowe funkcje, a przy okazji można wracać do poprzednich etapów.

- **Jak się testuje?**

Testowanie następuje po każdej iteracji, więc błędy wykrywa się wcześnie i na bieżąco. Pozwala to na ciągłe ulepszanie produktu.

- **Plusy:** Elastyczność i możliwość szybkiego reagowania na zmiany.
- **Minusy:** Może być drogie i czasochłonne, jeśli spirale trwają zbyt długo.

6. „Wielki wybuch”

W tym podejściu nie ma prawie żadnego planowania. Programiści po prostu zaczynają kodować, a produkt powstaje „na żywioł”. Najczęściej stosowane w małych projektach lub startupach.

- **Jak się testuje?**

Testowanie jest bardzo chaotyczne i często improwizowane. Bywa, że testy są pomijane albo przeprowadzane na samym końcu, co może prowadzić do licznych problemów w działaniu.

- **Plusy:** Szybki start projektu.
- **Minusy:** Wysokie ryzyko błędów i braku spójności w produkcie końcowym.

Podsumowując:

- Wybór cyklu życia ma kluczowy wpływ na sposób testowania.
- Modele bardziej uporządkowane (np. kaskadowy, spiralny) mają zaplanowane testy w konkretnych etapach.
- Modele elastyczne (prototypowanie, spiralny) pozwalają na testowanie na bieżąco.
- Chaotyczne podejścia („Wielki wybuch”) często zaniedbują testowanie, co prowadzi do późniejszych problemów.

Metody pomiaru jakości systemu informatycznego

Pomiar jakości systemu polega na zastosowaniu różnych metod, które pozwalają ocenić, czy system działa zgodnie z wymaganiami, jest niezawodny, wydajny i spełnia oczekiwania użytkowników. Oto omówienie głównych metod:

1. Przeglądy systemu

Regularne sprawdzanie systemu pod kątem zgodności z wymaganiami i standardami jakości.

- **Cel:** Zidentyfikowanie potencjalnych problemów na wczesnym etapie.
- **Kiedy się stosuje?** Na różnych etapach, od projektowania po implementację.

2. Nadzory nad systemem

Stałe monitorowanie działania systemu w celu identyfikacji odchyleń od oczekiwanych wyników.

- **Cel:** Zapewnienie ciągłej wydajności i minimalizacja ryzyka awarii.
- **Kiedy się stosuje?** W trakcie eksploatacji systemu.

3. Testowanie

Najbardziej powszechna metoda, polegająca na eksperymentowaniu z systemem, by znaleźć błędy i ocenić jakość.

- **Elementy testowania:**
    - **Eksperyment:** Celowe działanie mające na celu ocenę systemu.
    - **System informatyczny lub komponent:** Testowany obiekt.
    - **Dane wejściowe:** Przygotowane przypadki testowe.
    - **Przewidywane wyniki:** Oczekiwane rezultaty testu.
    - **Porównanie:** Analiza rzeczywistych wyników z oczekiwanymi.
- **Cele testowania:**
    - Znajdowanie błędów.
    - Ocena zgodności ze specyfikacją.

4. Audity systemu

Formalna i niezależna ocena jakości systemu, zazwyczaj przeprowadzana przez zewnętrznych specjalistów.

- **Cel:** Sprawdzenie, czy system spełnia określone standardy i wymagania.
- **Kiedy się stosuje?** Na końcowych etapach projektu lub podczas certyfikacji.

5. Inspekcje

Szczegółowe i systematyczne przeglądy kodu, dokumentacji lub procesów.

- **Cel:** Znalezienie błędów i problemów bez konieczności uruchamiania systemu.
- **Kiedy się stosuje?** Przed testowaniem lub równolegle z nim.

6. Inne metody

- **Symulacje:** Tworzenie warunków testowych, które naśladują rzeczywiste środowisko pracy.
- **Analiza ryzyka:** Ocena potencjalnych zagrożeń i ich wpływu na system.

Testowanie: Szczegółowe omówienie

Testowanie jako eksperyment

- **Definicja:** Testowanie to eksperyment polegający na sprawdzaniu, czy system działa poprawnie w określonych warunkach.
- **Elementy:**
  - **Eksperyment.**
  - **System lub komponent.**
  - **Dane wejściowe.**
  - **Przewidywane wyniki.**
  - **Porównanie wyników.**

Weryfikacja a Walidacja

- **Weryfikacja:** Czy robimy rzeczy dobrze?
  - Testowanie na danym etapie, by sprawdzić zgodność z wymaganiami fazy.
- **Walidacja:** Czy robimy właściwą rzecz?
  - Sprawdzenie gotowego produktu pod kątem zgodności z wymaganiami użytkownika.

Znajdowanie błędów

- Błąd, defekt, awaria, usterka, niezgodność itd.
- Błąd oprogramowania:
  - Niezgodność ze specyfikacją,
    - Nie wykonuje elementu specyfikacji,
    - Robi coś czego nie powinno,
  - Brak w specyfikacji pożądanej przez użytkownika funkcji,
  - Program jest trudny do użycia lub zrozumienia.

Definicje – błędu:

- **błąd:** zdarzenie w wyniku którego system produkuje nieoczekiwany rezultat lub zachowuje się w nieoczekiwany sposób
- **defekt:** nieprawidłowe działanie człowieka np. w trakcie kodowania lub podejmowania decyzji projektowych będące przyczyną błędu
- **awaria:** stan, w którym program nie jest zdolny wypełniać co najmniej jednej ze swoich funkcji

Rodzaje błędów (wg Beizera)

Klasa błędu	*Możliwy błąd*
Funkcjonalny	Zła lub brakująca funkcja
Systemowy	Błędnie użyte interfejsy, złe zarządzanie zasobami, zły przepływ sterowania
Przetwarzania	Niewłaściwe przetwarzanie danych w module
Danych	Błędna specyfikacja, projekt, rozmieszczenie lub inicjacja danych
Kodowania	Niewłaściwe użycie instrukcji języka programowania
Dokumentacyjny	Niepełna lub błędna treść dokumentacji
Inny	Przyczyny nieznane

Koszt naprawy błędów

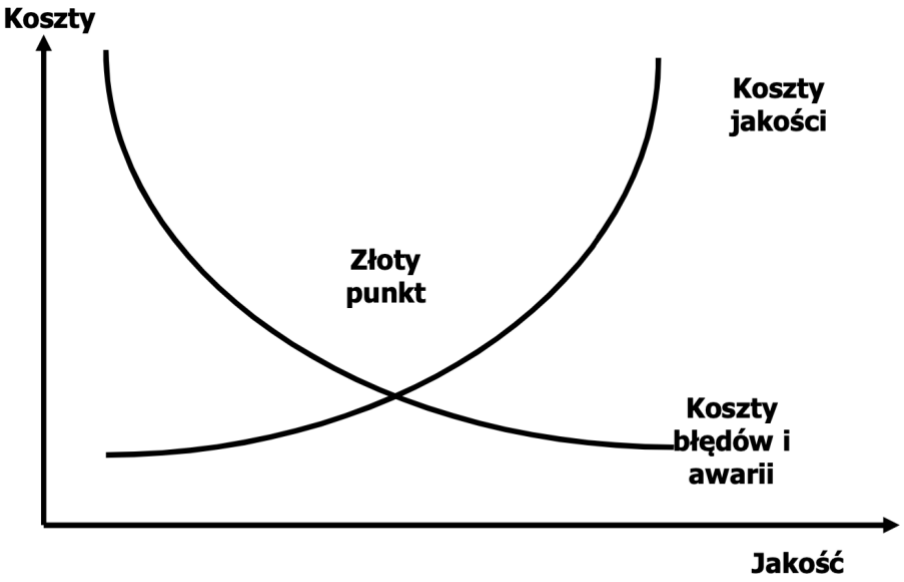
Błędy krytyczne i inne problemy w systemach informatycznych mogą generować ogromne straty finansowe. Przykładowo, w badaniach francuskich przedsiębiorstw:

- **Błędy krytyczne:** 3,5 mld PLN.

- **Inne błędy:** 5,275 mld PLN.

Problemy kosztów jakości

1. **Koszty jakości** – związane z zapewnieniem jakości (testowanie, audity).
2. **Koszty błędów** – wynikające z awarii i defektów.



Wniosek:

- Istnieje „złoty punkt” – balans między kosztami testowania a ryzykiem błędów.
- Więcej testowania ≠ lepsza jakość, ale pozwala na redukcję kosztów związanych z późniejszymi awariami.

Podział procesów testowania

a) Ze względu na strategię testowania:

- **Testowanie wstępujące**: zaczyna się od mniejszych części (podsystemów), a na końcu testuje się cały system.
- **Testowanie zstępujące**: startuje się od głównego modułu (całego systemu) i stopniowo dołącza się podsystemy – tzw. najpierw „szkielet”, później reszta.

b) Ze względu na metodę testowania:

- **Metoda szklanego pudełka (white box)**: testujący zna wnętrze programu (logikę, kod, strukturę), więc testuje się działanie „od środka”.
- **Metoda czarnej skrzynki (black box)**: testujący nie zna kodu ani wewnętrznej budowy, sprawdza głównie dane wejściowe i wyjściowe.
- **Metoda szarej skrzynki (gray box)**: częściowo zna się wnętrze, często używana np. przy testowaniu stron WWW.

c) W zależności od osób testujących:

- **Testowanie przez autora**: programista sam sprawdza swój kod.
- **Testowanie przez eksperta**: inna osoba, doświadczona w testach.
- **Testowanie przez użytkownika**: np. wersje beta oprogramowania udostępniane do testów.

d) Ze względu na cel przeprowadzenia testów:

- testowanie funkcjonalne, gdzie celem sprawdzenie poprawności realizacji funkcji systemu,
- testowanie regeneracji, gdzie „celem jest upewnienie się, że system można zregenerować po rozmaitych awariach”,
- testowanie efektywności, gdzie „celem jest upewnienie się, że system może obsłużyć zbiory danych o wymaganych rozmiarach i wszystkie przychodzące transakcje zgodnie z opisem w modelu implementacyjnym użytkownika, oraz że czas odpowiedzi jest zadowalający”.

e) Ze względu na podejście do testowania:

- **Pozytywne (delikatne)**: sprawdzanie funkcji, które powinny działać zgodnie z założeniami (tzw. „na spokojnie”).
- **Negatywne**: próba wymuszenia błędów i crashy – wprowadzasz dziwne dane, wywołujesz niestandardowe scenariusze.

f) Ze względu na obiekt testowy:

- **Testy modułów**: najmniejsze fragmenty systemu (np. pojedyncze klasy/funkcje).
- **Testy komponentów**: większe klocki, łączące kilka modułów.
- **Testy systemu**: testowanie całości aplikacji.
- **Testy odbioru**: ostateczne sprawdzenie, czy wszystko gra przed wdrożeniem u klienta.

Testowanie specyfikacji – zrozumiałe wprowadzenie

Testowanie specyfikacji to proces, który sprawdza, czy dokumentacja projektu (tzw. specyfikacja) jest kompletna, spójna i precyzyjna. Celem jest upewnienie się, że opis produktu – np. aplikacji, programu czy systemu – pozwala na jego prawidłowe stworzenie i późniejsze przetestowanie.

Co to jest specyfikacja?

Specyfikacja to dokument opisujący produkt za pomocą tekstu, ilustracji, schematów czy przykładów. Daje obraz tego, co ma być stworzone i jak powinno działać.

Przykład: Specyfikacja kalkulatora

- **Menu Edycja:** zawiera dwie funkcje: **Kopiuj** i **Wklej**.
- **Wybór funkcji:** użytkownik może je uruchomić na trzy sposoby:
  - Kliknięcie myszą.
  - Klawisze funkcyjne.
  - Standardowy skrót Windows (Ctrl+C i Ctrl+V).

Jak testuje się specyfikację?

1. **Metoda czarnej skrzynki**

- Skupia się na sprawdzeniu wymagań i funkcji bez wnikania w implementację.

2. **Praca badawcza**

- Obejmuje dokładne zrozumienie oczekiwań klienta, analizę branżowych standardów i reguł.
- Przykłady:
    - **Terminologia przedsiębiorstwa:** Czy użyte pojęcia są zgodne z przyjętymi definicjami?
    - **Standardy branżowe:** Czy kalkulator obsługuje skróty zgodnie z normami Windows?
    - **Graficzne interfejsy:** Czy design pasuje do systemu operacyjnego (np. Windows lub macOS)?
    - **Standardy sprzętowe:** Czy program działa na wymaganym sprzęcie?

Rola osób w testowaniu specyfikacji

- **Kierownik projektu:** Definiuje standardy i wymagania specyfikacji.
- **Tester:** Sprawdza, czy standardy są właściwe i czy specyfikacja zawiera wszystkie potrzebne informacje.
- **Porównanie z innymi programami:** Czy rozwiązanie odpowiada standardom rynku lub przewyższa konkurencję?

Techniki testowania specyfikacji

1. **Kontrolna lista atrybutów specyfikacji:**

Pomaga sprawdzić, czy specyfikacja jest:

- **Kompletna:** Czy opisano wszystkie wymagania?
- **Dokładna:** Czy są jakieś błędy w opisach?
- **Precyzyjna:** Czy opisy są jednoznaczne?
- **Spójna:** Czy różne elementy specyfikacji się nie wykluczają?
- **Istotna:** Czy wyeliminowano zbędne szczegóły?

- **Wykonalna:** Czy projekt można zrealizować w dostępnych zasobach?
- **Dająca się przetestować:** Czy wszystkie funkcje można jednoznacznie sprawdzić?

2. **Kontrolna lista terminologii:**

Pomaga unikać niejasnych sformułowań. Przykłady słów do weryfikacji:

- **Zbyt ogólne:** „Zawsze", „każdy", „nigdy" – mogą wprowadzać w błąd, bo nie są precyzyjne.
- **Niejasne:** „Dobry", „wydajny", „stabilny" – co dokładnie to oznacza?
- **Warunki:** „Jeśli… to" – co się stanie, jeśli warunek nie zostanie spełniony?
- **Synonimy:** „Podobne, adekwatne, analogiczne" – czy można je zastąpić bardziej jednoznacznymi określeniami?

Przykłady testów specyfikacji

1. **Kompletność:** Czy uwzględniono wszystkie funkcje kalkulatora?
   - Czy opisano, jak działa funkcja „Kopiuj" i „Wklej"?
   - Czy wspomniano o skrótach klawiszowych?
2. **Precyzja:** Czy specyfikacja jasno mówi, że skrót „Ctrl+V" działa tylko w systemie Windows?
3. **Spójność:** Czy interfejs użytkownika opisany w jednym miejscu specyfikacji jest zgodny z opisami w innych częściach dokumentu?

Dlaczego testowanie specyfikacji jest ważne?

- **Najtańsze wykrywanie błędów:** Błędy w specyfikacji można poprawić bez kosztów związanych z przebudową systemu.
- **Ułatwia implementację:** Dobrze przetestowana specyfikacja to jasne wytyczne dla programistów.
- **Redukuje ryzyko błędów:** Im lepiej opisana specyfikacja, tym mniejsze ryzyko, że produkt nie spełni oczekiwań klienta.

Co to jest Testowanie kodu?

Testowanie kodu to proces oceny jakości programu poprzez sprawdzanie jego działania na różnych poziomach. Ma na celu wykrycie błędów, zapewnienie zgodności z wymaganiami oraz zwiększenie niezawodności i wydajności systemu.

Dlaczego testowanie kodu jest ważne?

- **Wczesne wykrywanie błędów:** Im wcześniej znajdziemy błąd, tym łatwiej i taniej go naprawić.
- **Poprawa jakości:** Regularne testowanie zapewnia, że program działa zgodnie z oczekiwaniami.
- **Redukcja kosztów:** Zapobieganie błędom zmniejsza koszty związane z późniejszymi naprawami i wsparciem.

Poziomy Testowania kodu

Testowanie kodu odbywa się na różnych poziomach, z których każdy skupia się na innym aspekcie systemu:

a) Testowanie jednostkowe (Unit Testing)

- **Co to jest?** Testowanie najmniejszych fragmentów kodu, takich jak pojedyncze funkcje czy klasy.
- **Cel:** Sprawdzenie, czy poszczególne części programu działają poprawnie.
- **Kiedy?** W trakcie fazy kodowania.
- **Jak?** Autor kodu (programista) tworzy testy dla swoich jednostek, często automatycznie.

Przykład: Testowanie funkcji kalkulatora, która dodaje dwie liczby, aby upewnić się, że zwraca poprawny wynik.

b) Testowanie integracyjne (Integration Testing)

- **Co to jest?** Testowanie połączeń między różnymi jednostkami kodu.
- **Cel:** Upewnienie się, że moduły współpracują ze sobą prawidłowo.
- **Kiedy?** Po zakończeniu testowania jednostkowego.

Strategie testowania integracyjnego:

1. **Przyrostowe:**
   - **Oddolne:** Testowanie zaczyna się od dolnych poziomów modułów i stopniowo przechodzi w górę.
   - **Odgórne:** Testowanie zaczyna się od głównych modułów i stopniowo integruje niższe poziomy.
2. **Skokowe:** Wszystkie moduły są testowane jednocześnie, co jest przydatne w systemach z wieloma identycznymi komponentami.

Przykład: Testowanie interakcji między modułami logowania i rejestracji użytkowników.

c) Testowanie systemowe (System Testing)

- **Co to jest?** Testowanie całego systemu jako jednego, spójnego produktu.
- **Cel:** Sprawdzenie, czy system spełnia wszystkie wymagania funkcjonalne i niefunkcjonalne.
- **Kiedy?** Po zakończeniu testów integracyjnych.

Kategorie testowania systemowego:

- **Użyteczności:** Czy interfejs użytkownika jest intuicyjny?
- **Objętości:** Jak system radzi sobie z dużymi ilościami danych?
- **Zmęczeniowe:** Czy system działa stabilnie pod długotrwałym obciążeniem?
- **Poufności:** Czy system jest odporny na próby włamania?
- **Osiągów:** Jak szybko system reaguje na zapytania?

- **Pamięci:** Jak efektywnie system zarządza pamięcią?
- **Kompatybilności:** Czy system działa na różnych platformach?
- **Instalacji:** Jak łatwo jest zainstalować system?
- **Niezawodności:** Jak często system się psuje?
- **Odtwarzania:** Czy system może się zregenerować po awarii?
- **Obsługi:** Czy personel potrafi efektywnie zarządzać systemem?
- **Dokumentacji:** Czy dokumentacja jest kompletna i pomocna?

Przykład: Sprawdzenie, czy system bankowy prawidłowo przetwarza duże ilości transakcji jednocześnie.

d) Testowanie niezależne (Independent Testing)

- **Co to jest?** Testowanie przeprowadzane przez osoby, które nie są zaangażowane w tworzenie kodu.
- **Cel:** Zapewnienie obiektywnej oceny jakości systemu.
- **Kiedy?** Może odbywać się na różnych etapach rozwoju oprogramowania.

Przykład: Zespół QA (Quality Assurance) testuje aplikację mobilną stworzoną przez inny zespół programistów.

e) Testowanie akceptacyjne (Acceptance Testing)

- **Co to jest?** Ostateczne testy przeprowadzane przez użytkowników końcowych lub klientów.
- **Cel:** Upewnienie się, że system spełnia wszystkie wymagania użytkowników i jest gotowy do wdrożenia.
- **Kiedy?** Tuż przed wdrożeniem systemu do produkcji.

Przykład: Firma klient testuje nową wersję oprogramowania CRM przed jej oficjalnym wdrożeniem.

f) Testowanie alfa (Alpha Testing)

- **Co to jest?** Testowanie wersji beta oprogramowania wewnętrznie, przez zespół deweloperski.
- **Cel:** Wykrycie i naprawa błędów przed udostępnieniem produktu szerszej grupie użytkowników.
- **Kiedy?** W fazie przed oficjalnym wydaniem produktu.

Przykład: Programiści testują nową funkcję czatu w aplikacji społecznościowej przed jej publicznym uruchomieniem.

g) Testowanie beta (Beta Testing)

- **Co to jest?** Testowanie wersji beta oprogramowania przez ograniczoną grupę użytkowników zewnętrznych.
- **Cel:** Zbieranie opinii i wykrywanie błędów w rzeczywistych warunkach użytkowania.
- **Kiedy?** Po testach alfa, przed pełnym wdrożeniem.

Przykład: Firma udostępnia wersję beta nowej gry komputerowej wybranym graczom do testowania.

Testowanie jednostkowe – Szczegółowe Omówienie

Testowanie jednostkowe koncentruje się na najmniejszych częściach kodu – funkcjach, metodach lub klasach. Jest to pierwszy poziom testowania, który pomaga wykryć błędy na wczesnym etapie.

Kluczowe Elementy Testowania jednostkowego:

- **Wyodrębnione fragmenty kodu:** Testuje się pojedyncze jednostki, niezależnie od reszty systemu.
- **Oderwanie od reszty kodu:** Dzięki izolacji, testy są szybsze i bardziej niezawodne.
- **Fazy testowania:**
    1. **Faza 1 – Autor:** Programista pisze testy dla swoich jednostek.
    2. **Faza 2 – Autor + Zespół uruchomieniowy:** Testy są sprawdzane i uruchamiane przez zespół.

Dodatkowe Elementy:

- **Testowanie regresywne:** Powtarzanie testów po wprowadzeniu zmian w kodzie, aby upewnić się, że nowe zmiany nie wprowadziły nowych błędów.
- **Zestawy testowe:** Kolekcje testów, które mogą być używane wielokrotnie.
- **Automaty testujące:** Narzędzia automatyzujące proces testowania, co przyspiesza i ułatwia pracę.

Przykład: Testowanie funkcji obliczającej średnią arytmetyczną, aby upewnić się, że zwraca prawidłowe wyniki dla różnych zestawów danych.

Testowanie integracyjne – Szczegółowe Omówienie

Testowanie integracyjne sprawdza, czy różne moduły lub komponenty systemu współpracują ze sobą prawidłowo.

Strategie testowania integracyjnego:

a) Przyrostowe

- **Opis:** System jest budowany stopniowo, a każdy nowy komponent jest testowany wraz z już istniejącymi.
- **Zalety:** Ułatwia identyfikację błędów na wczesnym etapie.

b) Skokowe

- **Opis:** Wszystkie moduły są łączone i testowane jednocześnie.
- **Zalety:** Szybsze, ale trudniejsze do zarządzania w dużych systemach.

Podejścia testowania przyrostowego:

i) Oddolne (Bottom-Up)

- **Opis:** Testowanie zaczyna się od najniższych poziomów modułów i stopniowo przechodzi w górę.
- **Zalety:** Łatwiejsze projektowanie testów i tworzenie sterowników.

- **Preferowane przez:** Programistów.

ii) Odgórne (Top-Down)

- **Opis:** Testowanie zaczyna się od najwyższych poziomów modułów i stopniowo integruje niższe poziomy.
- **Zalety:** Preferowane przez użytkowników końcowych, ale trudniejsze do implementacji.
- **Preferowane przez:** Użytkowników końcowych.

Przykład Testowania integracyjnego:

- **Oddolne:** Najpierw testujemy funkcje bazowe, a następnie integrujemy je z bardziej zaawansowanymi funkcjami.
- **Odgórne:** Najpierw testujemy główną funkcjonalność systemu, a potem dodajemy bardziej szczegółowe moduły.

Testowanie systemowe – Szczegółowe Omówienie

Testowanie systemowe polega na sprawdzeniu całego systemu jako jednego, spójnego produktu, aby upewnić się, że spełnia wszystkie wymagania.

Kluczowe Elementy Testowania systemowego:

- **Skoncentrowane na całości:** Testujemy system jako jeden duży blok, a nie jego poszczególne części.
- **Wybór testów:** Oparta na szczegółowej specyfikacji wymagań.
- **Dane dla walidacji:** Wyniki testów systemowych służą do walidacji, czy system spełnia oczekiwania użytkowników.

Typowe Kategorie Testowania systemowego:

Test	*Czynność*	*Zastosowanie*
użyteczności	sprawdzenie każdego "co robi"	rozbudowana funkcjonalność
objętości	olbrzymie dane wejściowe	nieograniczone wejście
zmęczeniowy	intensywne dane wejściowe	systemy RT
obsługi	sprawdzenie przyjazności	systemy interakcyjne
poufności	próba włamania	systemy z poufnością
osiągów	pomiar parametrów	systemy RT
pamięci	pomiar zajętości pamięci	krytyczne zapotrz. na pamięć
konfiguracji	próba pracy w różnych środowiskach	oprogr.rozszerzalne
kompatybilności	testowanie po rozbudowie	nowe wersje
instalacji	testowanie procedur instalacyjnych	skomplikowana instalacja
niezawodności	rejestracja danych o niezawodności np. MTBF	wymagana niezawodność
odtwarzania	symulowanie uszkodzeń sprzętu	systemy FT
obsługi	przećwiczenie czynności obsługowych	systemy administrowane
dokumentacji	sprawdzenie przydatności dokumentacji	systemy administrowane
proceduralny	sprawdzenie procedur wykonywanych przez ludzi	wspieranie decyzji i dowodzenie

Przykład: Testowanie systemu bankowego pod kątem użyteczności, wydajności i bezpieczeństwa przed jego wdrożeniem.

Podsumowanie

Testowanie kodu jest niezbędnym procesem, który zapewnia, że oprogramowanie działa zgodnie z założeniami i spełnia wymagania użytkowników. Dzięki różnym poziomom i metodom testowania możemy dokładnie sprawdzić każdy aspekt systemu, od najmniejszych jednostek kodu po całe, zintegrowane środowisko.

Kluczowe Wnioski:

- **Różnorodność testów:** Każdy poziom testowania ma swoje specyficzne cele i metody.
- **Wczesne wykrywanie błędów:** Testowanie jednostkowe i integracyjne pomagają znaleźć błędy na wczesnym etapie.
- **Kompleksowe podejście:** Testowanie systemowe obejmuje wszystkie aspekty działania systemu, zapewniając jego niezawodność i wydajność.
- **Współpraca zespołów:** Testowanie wymaga zaangażowania różnych osób – od programistów po testerów i użytkowników końcowych.

Testowanie kodu to fundament solidnego i niezawodnego oprogramowania. Inwestując czas i zasoby w odpowiednie testowanie, zapewniasz sukces swojego projektu i zadowolenie użytkowników.

Kolejna klasyfikacja testowania

Rodzaje testów ze względu na cel

- **Wykrywanie błędów**: Celem jest odnalezienie błędów, które mogą wpłynąć na działanie programu.
  - Przykład: Testowanie funkcji kalkulatora, by upewnić się, że poprawnie wykonuje operacje arytmetyczne.
- **Testy statystyczne**: Analiza danych, która pozwala wykryć przyczyny najczęstszych błędów oraz ocenić niezawodność systemu.
  - Przykład: Zbadanie, dlaczego użytkownicy najczęściej napotykają na problemy w określonym scenariuszu.

Rodzaje testów ze względu na technikę wykonywania

- **Dynamiczne**: Polegają na uruchamianiu programu lub jego fragmentów i porównywaniu wyników z oczekiwanymi.
  - Przykład: Testowanie, czy aplikacja poprawnie reaguje na kliknięcie przycisku.

- **Statyczne**: Analiza kodu bez uruchamiania programu.
  - Przykład: Przegląd kodu w celu znalezienia błędów składniowych lub logicznych.

Testy statyczne

Testy statyczne koncentrują się na kodzie, a nie na działaniu programu.

Techniki testów statycznych:

1. **Dowody poprawności**: Formalne metody matematyczne, które udowadniają, że kod działa poprawnie zgodnie ze specyfikacją.
   - **Wady:** Wysoki koszt i możliwość popełnienia błędów w dowodach. Rzadko stosowane w praktyce.
2. **Metody nieformalne**:
   - **Śledzenie przebiegu programu:** Analiza kodu krok po kroku w celu znalezienia błędów.
   - **Wyszukiwanie typowych błędów:** Przeglądanie kodu, by znaleźć znane problemy, takie jak złe zarządzanie pamięcią.

Ocena liczby błędów

Koszt konserwacji oprogramowania zależy od:

- Liczby błędów w programie.
- Procentu błędów zgłaszanych przez użytkowników.
- Kosztu usunięcia błędów (na podstawie doświadczeń z poprzednich projektów).

Testowanie strukturalne

Testowanie strukturalne opiera się na znajomości kodu i analizie jego działania.

Rodzaje testów strukturalnych:

1. **Testowanie rozgałęzień i ścieżek**

   • **Rozgałęzienia:** Dane testowe powodują przejście przez każdą ścieżkę programu.

   • **Ścieżki:** Dane testowe sprawdzają wszystkie możliwe kombinacje ścieżek.

Przykład (instrukcja warunkowa):

```
if (x > 10) {
  console.log("Większe niż 10");
} else {
  console.log("Mniejsze lub równe 10");
}
```

- **Rozgałęzienia:** Testujemy obie ścieżki: x > 10 i x <= 10.
- **Ścieżki:** Rozważamy każdą kombinację ścieżek w większym fragmencie kodu.

2. **Testowanie pojedynczych instrukcji i przepływu danych**

   • Analiza tego, jak dane są przesyłane między różnymi fragmentami kodu.

3. **Analiza mutacyjna**

   • Tworzenie zmodyfikowanych wersji kodu (mutacji) i sprawdzanie, czy testy potrafią wykryć wprowadzone błędy.

Strategie testowania

Metoda rozgałęzień i ścieżek

- **Założenie:** Źródłem błędów są problemy w przepływie sterowania.
- **Cel:** Upewnienie się, że program działa poprawnie w każdej możliwej sytuacji.

Przykład (ścieżki):
Dla prostego diagramu:

1 → 2 → 4 → 6
1 → 3 → 5 → 6

Rozgałęzienia:

- Testujemy każdą pojedynczą ścieżkę: 1-2-4-6 oraz 1-3-5-6.

Ścieżki:

- Testujemy wszystkie kombinacje: 1-2-4-6, 1-3-4-6, 1-2-5-6, 1-3-5-6.

Metoda zasiewania błędów

- **Opis:** Celowe wprowadzanie błędów do programu, aby sprawdzić skuteczność testów w ich wykrywaniu.
- **Proces:**
    1. Wprowadzamy określoną liczbę błędów do programu.
    2. Inna osoba lub zespół wykonuje testy.
    3. Porównujemy liczbę wykrytych błędów z rzeczywistą liczbą błędów wprowadzonych.

Formuła:

- **Przed testami:**
- **Po testach:**
    - **N:** liczba wprowadzonych błędów.
    - **M:** liczba wszystkich wykrytych błędów.
    - **X:** liczba wprowadzonych błędów, które zostały wykryte.

Zalety: Pozwala ocenić skuteczność metody testowej i przewidzieć liczbę niewykrytych błędów.

Przeglądy i inspekcje kodu

- **Przeglądy kodu:** Grupa programistów analizuje kod, szukając błędów lub obszarów do poprawy.
- **Inspekcje kodu:** Formalny proces, gdzie zespół ocenia kod według określonych kryteriów.

Przykład:
Podczas przeglądu programista może zauważyć brak obsługi błędów w funkcji logowania.

Podsumowanie

- **Statyczne testy** pomagają znaleźć błędy w kodzie bez jego uruchamiania. Są szybsze, ale mniej dokładne w wykrywaniu błędów dynamicznych.
- **Strukturalne testy** badają przepływ sterowania i dane w programie, zapewniając lepsze zrozumienie działania kodu.
- **Zasiewanie błędów** to technika, która pozwala ocenić skuteczność testów i przewidzieć liczbę niewykrytych problemów.
- **Przeglądy i inspekcje kodu** to kluczowe elementy procesu testowania, które pomagają w poprawie jakości oprogramowania.

Dobór danych testowych

Dobór odpowiednich danych testowych jest kluczowym etapem testowania oprogramowania. To dzięki nim możemy skutecznie wykryć błędy i upewnić się, że system działa poprawnie w różnych sytuacjach.

1. Kryteria doboru danych testowych

a) Kryterium pokrycia wszystkich funkcji

- **Opis:** Dane wejściowe dobieramy tak, aby każda funkcja programu została wykonana przynajmniej raz.
- **Zalety:**
  - Niska liczba wymaganych testów.
  - Proste w implementacji.
- **Wady:**
  - Może być nieskuteczne – nie gwarantuje pokrycia wszystkich możliwych przypadków.

b) Kryterium pokrycia instrukcji warunkowych

- **Opis:** Dane wejściowe muszą zapewnić, że każdy elementarny warunek instrukcji warunkowej zostanie przetestowany zarówno jako spełniony, jak i niespełniony.
- **Przykład:**

```
if (x > 5) {
  console.log("Większe niż 5");
} else {
  console.log("Mniejsze lub równe 5");
}
```

Dane testowe:

- (warunek spełniony).
- (warunek niespełniony).

2. Strategie testowania – metoda analizy mutacyjnej

Metoda analizy mutacyjnej

- **Opis:**
  - Automatyczne generowanie „mutantów” programu – zmodyfikowanych wersji kodu.
  - Przykładowe zmiany w mutantach:
    - Zmiana nazw zmiennych.
    - Modyfikacja wartości literałów.
    - Zmiana kolejności argumentów w funkcjach.
    - Zmiana warunków logicznych.
- **Cel:**
  - Sprawdzić, czy dane testowe potrafią wykryć wprowadzone celowo błędy.
  - „Zabicie mutanta” oznacza, że dane testowe skutecznie wykryły różnicę między programem oryginalnym a mutantem.

Przykład:
W programie, który sprawdza liczby pierwsze, mutant może mieć zmieniony warunek:

```
if (n % 2 === 0 && n !== 2) { // Mutant zmienia warunek
  return false;
}
```

- **Efekty:**
  - Skuteczność testów można ocenić na podstawie liczby zabitych mutantów.
  - System automatycznego testowania mutacyjnego (np. narzędzia XP) pomaga w analizie wyników.

3. Programowanie i testowanie ekstremalne (XP)

Metodologia **XP (Extreme Programming)** jest przeznaczona dla zespołów pracujących w projektach z szybko zmieniającymi się wymaganiami.

Zasady XP:

1. **Określenie potrzeb:** Programiści spotykają się z klientem, aby zrozumieć jego potrzeby i stworzyć historie użytkownika.
2. **Planowanie zadań:** Programiści ustalają, jakie zadania muszą być wykonane i dzielą się nimi.
3. **Harmonogram:** Klient priorytetyzuje zadania, a zespół tworzy harmonogram.
4. **Programowanie w parach:** Każde zadanie jest realizowane przez dwóch programistów.
5. **Środowisko testowe:** Każda para tworzy testy na podstawie specyfikacji.
6. **Ciągłe testowanie i poprawki:** Testy są przeprowadzane cyklicznie, aż kod przejdzie wszystkie testy.
7. **Ciągła integracja:** Codzienna synchronizacja kodu z bazowym repozytorium.

8. **Wersja przedprodukcyjna:** Zespół przekazuje klientowi wstępną wersję aplikacji.

9. **Testy akceptacyjne:** Klient przeprowadza testy, aby upewnić się, że system spełnia jego wymagania.

10. **Wersja końcowa:** Gotowy produkt jest dostarczany klientowi.

4. Techniki kodowania w XP

- **Proste projektowanie i kodowanie:** Skupienie na prostocie, bez zbędnego komplikowania kodu.
- **Bezlitosna refaktoryzacja:** Regularne ulepszanie kodu w celu utrzymania jego jakości.
- **Standardy kodowania:** Zespół stosuje wspólne zasady, aby kod był spójny.
- **Wspólne słownictwo:** Wszyscy używają tych samych terminów, aby uniknąć nieporozumień.

5. Techniki tworzenia w XP

- **Kreowanie nakierowane na testy:** Testy są tworzone przed implementacją kodu.
- **Programowanie w parach:** Dwie osoby pracują nad tym samym fragmentem kodu, co pozwala na szybsze wykrywanie błędów.
- **Kolektywna własność kodu:** Każdy programista może wprowadzać zmiany w dowolnym fragmencie kodu.
- **Ciągła integracja:** Kod jest regularnie synchronizowany i testowany.

6. Przykład aplikacji do testowania

Opis aplikacji:

- Aplikacja wczytuje liczbę całkowitą w zakresie .
- Sprawdza, czy liczba jest pierwsza.
- Generuje odpowiednie komunikaty w zależności od wyniku.

Dane testowe:

1. **Poprawne dane w zakresie:**
   - (liczba pierwsza).
   - (nie jest liczbą pierwszą).
2. **Granice zakresu:**
   - i (wartości graniczne).
3. **Niepoprawne dane:**
   - (liczba spoza zakresu).
   - (liczba spoza zakresu).
   - (nie jest liczbą).

Oczekiwane wyniki:

- : „Liczba jest pierwsza”.
- : „Liczba nie jest pierwsza”.
- : „Podaj liczbę w zakresie od 0 do 1000”.
- : „Podaj poprawną liczbę całkowitą”.

Podsumowanie

Dobór danych testowych to kluczowy proces w testowaniu oprogramowania, który:

- Gwarantuje pokrycie wszystkich funkcji i warunków w kodzie.
- Pozwala na wykrycie błędów logicznych i granicznych.
- Wspiera skuteczność testów poprzez zaawansowane techniki, takie jak analiza mutacyjna.

Inne metody testowania

W testowaniu oprogramowania istnieje wiele metod i narzędzi wspomagających proces weryfikacji i walidacji kodu. Poniżej znajdziesz opis kluczowych technik i narzędzi, które pozwalają na bardziej zaawansowaną analizę działania programu.

1. Programy uruchamiające

Czym są?

Programy uruchamiające (debuggery) to narzędzia używane do wewnętrznego i zewnętrznego testowania. Wykorzystują zasadę białej skrzynki, co oznacza, że tester ma pełny dostęp do kodu i jego struktury.

Właściwości programów uruchamiających:

- **Wyświetlanie stanu zmiennych:** Możliwość podglądu wartości zmiennych w czasie działania programu.
- **Krokowe wykonywanie programu:** Uruchamianie kodu linia po linii w celu zrozumienia jego działania.
- **Punkty kontrolne (breakpoints):** Zatrzymanie programu w określonych miejscach, aby przeanalizować jego stan.
- **Obserwatorzy wartości zmiennych:** Monitorowanie wybranych zmiennych w czasie działania programu.
- **Zarządzanie kodem źródłowym:** Możliwość wprowadzania poprawek bezpośrednio w trakcie testowania.
- **Tworzenie dziennika testowania:** Rejestrowanie przebiegu testów, co pozwala na ich późniejsze odtworzenie.

Zastosowanie:

Przydatne do szczegółowej analizy kodu, diagnozowania błędów i poprawiania ich w locie.

2. Analizatory pokrycia kodu

Czym są?

Narzędzia, które sprawdzają, które fragmenty kodu zostały uruchomione podczas testów.

Korzyści:

- **Wykrycie kodu martwego:** Znalezienie części kodu, które nigdy nie są wykonywane.
- **Identyfikacja rzadko używanego kodu:** Analiza fragmentów kodu uruchamianych tylko przy specyficznych danych wejściowych.
- **Sumowanie wyników:** Możliwość zbierania danych z wielu przebiegów testów, aby uzyskać pełny obraz pokrycia.
- **Generowanie grafów sterowania:** Wizualizacja przepływu sterowania w programie.

Zastosowanie:

Idealne dla projektów wymagających pełnego pokrycia kodu testami, szczególnie w krytycznych systemach, takich jak oprogramowanie medyczne czy lotnicze.

3. Programy porównujące

Czym są?

Narzędzia umożliwiające porównanie dwóch wersji programu, plików lub wyników testów w celu identyfikacji różnic i wspólnych cech.

Zastosowanie:

- **Porównywanie wyników:** Analiza, czy wyniki testów zgadzają się z oczekiwaniami.
- **Testowanie interfejsów graficznych:** Pomocne w sprawdzaniu, czy GUI zachowuje się zgodnie z projektem.

Przykład:

Porównanie wyników działania starej i nowej wersji programu w celu upewnienia się, że nowe zmiany nie wprowadziły regresji.

4. Testy obciążeniowe i odpornościowe

a) Testy obciążeniowe

- **Cel:** Sprawdzenie wydajności i niezawodności systemu pod pełnym lub nadmiernym obciążeniem.
- **Zastosowanie:** Systemy wielodostępne, sieciowe lub aplikacje obsługujące wielu użytkowników jednocześnie.

Przykład:
Testowanie serwera bankowego przy 10 000 jednoczesnych transakcjach, aby upewnić się, że system nie ulegnie przeciążeniu.

b) Testy odpornościowe

- **Cel:** Sprawdzenie, jak system radzi sobie w sytuacjach awaryjnych, takich jak utrata połączenia sieciowego czy uszkodzenie sprzętu.
- **Zastosowanie:** Systemy krytyczne, które muszą działać nawet w ekstremalnych warunkach.

Przykład:
Symulowanie awarii zasilania podczas przetwarzania danych w systemie przemysłowym.

5. Testowanie konfiguracji

Cel:

Sprawdzenie, czy program działa poprawnie w różnych konfiguracjach sprzętowych i programowych.

Zastosowanie:

- **Przykład problemu:** Program działa na drukarkach laserowych, ale nie obsługuje drukarek atramentowych.
- **Scenariusze testowe:** Różne systemy operacyjne, wersje przeglądarek, typy urządzeń (np. desktop, tablet, telefon).

6. Testowanie kompatybilności

Cel:

Sprawdzenie, czy program współpracuje z innymi aplikacjami, systemami lub plikami danych.

Zastosowanie:

- **Przykład:**
  - Sprawdzenie, czy funkcja „Wklej” działa poprawnie przy przenoszeniu danych z arkusza kalkulacyjnego do programu tekstowego.
  - Testowanie importu i eksportu danych w aplikacji do hurtowni danych.

Podsumowanie

Inne metody testowania pozwalają na szczegółową analizę i diagnozę kodu oraz jego działania w różnych warunkach.

- **Programy uruchamiające:** Umożliwiają szczegółową analizę krokową.
- **Analizatory pokrycia kodu:** Pomagają w uzyskaniu pełnego pokrycia testami.
- **Programy porównujące:** Niezbędne przy testowaniu GUI lub wyników testów.
- **Testy obciążeniowe i odpornościowe:** Sprawdzają, jak system działa w ekstremalnych warunkach.
- **Testowanie konfiguracji i kompatybilności:** Upewniają się, że program działa na różnych urządzeniach i współpracuje z innymi aplikacjami.

Testy użytkowości i akceptacyjne

Testy użytkowości i akceptacyjne skupiają się na ocenie, czy system spełnia wymagania użytkowników i czy może być wdrożony jako gotowy produkt. Oto, jak wyglądają te procesy krok po kroku.

1. Testowanie akceptacyjne

Czym jest?

- To testowanie przeprowadzane **z udziałem klienta**, który ma zaakceptować system jako gotowy do wdrożenia.

- Ma charakter **“demonstracji”**, a nie szczegółowego badania.

Elementy testowania akceptacyjnego:

1. **Testowanie systemowe + klient:**

Testy sprawdzają funkcje i działanie całego systemu z punktu widzenia użytkownika końcowego.

2. **Rodzaje akceptacji:**

- **Fazowa:** Akceptacja etapowa, gdzie klient sprawdza poszczególne funkcjonalności lub moduły.
- **Ostateczna:** Końcowe zatwierdzenie systemu jako gotowego do wdrożenia.

Przykład: Klient testuje nowy system ERP, upewniając się, że spełnia wymagania biznesowe, takie jak wystawianie faktur czy zarządzanie zapasami.

2. Testowanie użytkowości

Czym jest?

Testowanie użytkowości odpowiada na pytanie:
„Czy system jest wystarczająco dobry, by zaspokoić potrzeby użytkowników i ich interesariuszy?”
(Za definicją Nielsena, 1993).

Główne cele:

- **Ocena przydatności:** Czy użytkownicy mogą osiągnąć swoje cele za pomocą systemu?
- **Ocena funkcjonalności:** Czy system działa zgodnie z założeniami i zapewnia wszystkie wymagane funkcje?

Charakterystyka testów użytkowości:

- **Metoda czarnej skrzynki:** Użytkownik nie musi znać wewnętrznej budowy systemu. Skupia się na działaniu i wynikach.
- **Praktyczne podejście:** Testy mają na celu sprawdzenie, jak dobrze system radzi sobie w rzeczywistych sytuacjach użytkowych.

3. Proces testowania użytkowości

Fazy procesu testowania (wg Fenta, 1991):

1. **Definicja zadania testowego:**

Określenie, co dokładnie chcemy przetestować i jakie są cele testu.

2. **Określenie obiektu testowania:**

Wybór funkcji lub modułów, które będą testowane.

3. **Przeprowadzenie pomiarów:**

Realizacja testów zgodnie z założonym planem.

4. **Analiza wyników:**

Zbieranie i przetwarzanie danych z testów.

5. **Ocena wyników:**

Porównanie wyników rzeczywistych z oczekiwanymi.

6. **Podjęcie decyzji:**

Określenie, czy system spełnia wymagania, czy potrzebne są poprawki.

7. **Dokonanie poprawek:**

Wprowadzenie zmian na podstawie wyników testów i decyzji.

4. Plan testowania użytkowości

Elementy planu testowego:

1. **Obiekt testowania:** Co będzie testowane (np. funkcja wyszukiwania w aplikacji).
2. **Miejsce i termin:** Gdzie i kiedy przeprowadzone zostaną testy.
3. **Uczestnicy:** Osoby odpowiedzialne za testy (testerzy, użytkownicy).

Opis testu:

- **Dane wejściowe:** Jakie dane zostaną wprowadzone do systemu.
- **Oczekiwane wyniki:** Jakie wyniki są oczekiwane po wykonaniu testu.

Procedury testowe:

- **Przygotowanie i wprowadzenie danych:** Jak należy wprowadzić dane do systemu.
- **Zachowanie wyników:** Jak przechowywać wyniki testów.
- **Analiza wyników:** Jak analizować wyniki w celu wyciągnięcia wniosków.

5. Kluczowe pytania i cele testów użytkowości

- **Czy system jest wystarczająco dobry?**

Ocenia się, czy użytkownicy mogą skutecznie korzystać z systemu w celu realizacji swoich zadań.

- **Czy system jest funkcjonalny?**

Sprawdza się, czy system oferuje wszystkie funkcje określone w specyfikacji.

6. Podsumowanie

Testy akceptacyjne i **testy użytkowości** skupiają się na ocenie systemu z perspektywy użytkownika końcowego.

- **Testy akceptacyjne:** Są ostatnim krokiem przed wdrożeniem systemu i angażują klienta.
- **Testy użytkowości:** Sprawdzają, czy system spełnia potrzeby użytkowników i pozwala im osiągnąć cele.

Różnice:

- **Testy akceptacyjne** to demonstracja funkcjonalności systemu dla klienta.
- **Testy użytkowości** to bardziej szczegółowa ocena użyteczności i przydatności systemu.

Przydatność systemu

Przydatność systemu odnosi się do zdolności systemu do spełnienia wymagań użytkowników i realizacji potrzebnych funkcji. Testowanie przydatności pomaga ocenić, czy system informatyczny dostarcza oczekiwanych korzyści i czy wymaga modernizacji lub popraw.

1. Czym jest przydatność systemu?

Przydatność systemu to:

- Stopień, w jakim system realizuje zdefiniowane funkcje zgodnie z potrzebami użytkownika.
- Odpowiedź na pytania:
  - Czy system potrafi wykonać zadane funkcje?
  - Czy funkcje, które oferuje, są tymi, które są potrzebne?

Zasada: Im mniej funkcji system realizuje z zakresu wymaganych, tym niższa jest jego przydatność.

2. Przykładowe stopnie przydatności systemów

a) Systemy informatyczne przedsiębiorstw

- Przydatność oceniana na podstawie zdolności do realizacji wymaganych funkcji organizacyjnych, takich jak księgowość, zarządzanie zasobami czy raportowanie.

b) Programy edukacyjne

- Przydatność zależy od tego, ile i czego użytkownicy mogą się nauczyć za pomocą programu.

c) Gry komputerowe

- Oceniana pod kątem zdolności dostarczania rozrywki użytkownikom.

3. Metoda testowania przydatności

Podstawy:

- Testowanie polega na znajomości funkcji, które system powinien realizować, oraz sprawdzeniu, czy wykonuje je poprawnie.

Etapy testowania:

1. **Wywiady z użytkownikami:** Ustalanie, czy system spełnia ich oczekiwania.
2. **Obserwacje:** Analiza sposobu użytkowania systemu w rzeczywistych warunkach.
3. **Kwestionariusze:** Zbieranie opinii i informacji od użytkowników w ustrukturyzowanej formie.

4. Informacje o realizacji funkcji

Skąd pochodzi informacja o problemach z funkcjami?

1. **Uwagi użytkowników:**
   - Zgłoszenia błędów lub braków w funkcjach.
2. **Wcześniejsza dokumentacja:**
   - Analiza istniejących specyfikacji lub opisów systemu.

5. Proces testowania przydatności

Testowanie przydatności odpowiada na pytanie:
„Czy system realizuje wymagane funkcje i czy są one wykonane w odpowiedni sposób?”

Kto przeprowadza testy?

- **Analityk:** Określa wymagania i sprawdza ich realizację.
- **Projektant:** Weryfikuje, czy zaprojektowane funkcje działają zgodnie z założeniami.
- **Użytkownik:** Oceni, czy system spełnia jego potrzeby.

Kiedy przeprowadza się testy?

1. **Okresowo:**
   - W celu globalnej oceny przydatności systemu.
   - Sprawdzenie, czy system nadal odpowiada potrzebom użytkowników, czy wymaga modernizacji.
2. **Jednorazowo:**
   - Po aktualizacjach systemu lub zgłoszeniach użytkowników dotyczących zmian.

6. Formularz do zapisu testów

Elementy formularza:

- **Lista funkcji:** Jakie funkcje powinien realizować system?
- **Możliwość realizacji:** Czy dana funkcja jest dostępna w systemie?
- **Sposób realizacji:** Czy realizacja funkcji jest zgodna z potrzebami użytkownika?
- **Uwagi:** Dodatkowe informacje o problemach lub brakach.

Przykład formularza:

Funkcja systemu	**Możliwość realizacji przez system**	**Odpowie-dni sposób realizacji**	Uwagi
drukowanie raportu kasowego	+	+	**brak**
zapis planu kont w bazie danych	-	-	**brak**
wystawianie dokumentów sprzedaży	+	-	**niewłaści we dane**

7. Kluczowe pytania podczas testów

1. Czy wszystkie wymagane funkcje są realizowane przez system?
2. Czy realizacja funkcji jest poprawna i spełnia potrzeby użytkowników?
3. Jakie są przyczyny niezrealizowania funkcji?

8. Źródła złej przydatności systemu

Główne przyczyny:

1. **Błędne założenia:**
   - Niewłaściwie określone wymagania użytkowników.
2. **Błędy projektowe:**
   - Pominięcie istotnych funkcji w procesie projektowania systemu.
3. **Błędy implementacyjne:**
   - Problemy wynikające z błędów w kodzie, które uniemożliwiają poprawne działanie funkcji.

Podsumowanie

Co sprawdza testowanie przydatności?

- Czy system realizuje wszystkie wymagane funkcje.
- Czy funkcje są wykonane w odpowiedni sposób.
- Jakie są źródła problemów w przypadku, gdy funkcje nie są realizowane.

Dlaczego jest to ważne?

Testowanie przydatności pozwala:

- Zidentyfikować luki w funkcjonalności systemu.
- Sprawdzić, czy system spełnia potrzeby użytkowników.
- Określić, czy potrzebne są modernizacje lub poprawki.

Funkcjonalność systemu – przegląd metod i technik

Definicja funkcjonalności systemu

- W jakim stopniu użytkownik może korzystać z funkcji systemu?
- Funkcjonalność systemu nie ocenia, czy system realizuje odpowiednie funkcje (to domena przydatności systemu).

Atrybuty funkcjonalności (wg Nielsen, 1993):

1. Łatwość nauki.
2. Wydajność.
3. Łatwość zapamiętywania.
4. Liczba i wpływ błędów.
5. Satysfakcja użytkownika.

Metody testowania funkcjonalności

- Testy funkcjonalności wykonane przez użytkowników.
- Testy pośrednie i bezpośrednie.

Testy funkcjonalności wykonane przez użytkowników

- **Cel:** Poprawa funkcjonalności testowanego produktu.
- **Metody:**
  - Użytkownicy wykonują rzeczywiste zadania.
  - Wszystkie działania i wypowiedzi są obserwowane i rejestrowane.
  - Analiza danych ujawnia problemy i sugeruje zmiany.

Metody testów funkcjonalności

1. Pomiar wyników pracy

- **Opis:** Testowa grupa użytkowników wykonuje zdefiniowany zestaw zadań, a badacze kolekcjonują dane takie jak czas, błędy i ich lokalizacja.
- **Dane:**
  - Wydajność użytkowania.
  - Analiza porównawcza alternatywnych systemów.
- **Zalety:** Konkretne wartości liczbowe, łatwe porównanie wyników.
- **Wady:** Brak identyfikacji indywidualnych problemów użytkowników.

2. Głośne myślenie

- **Opis:** Użytkownicy werbalizują swoje myśli podczas pracy z systemem.
- **Rodzaje:**
  - Konstruktywna interakcja (praca w parach).
  - Testowanie retrospektywne (analiza nagrań sesji testowych).
  - Kierowane myślenie (sterowanie działaniami użytkownika).
- **Zalety:** Wyławia błędne zrozumienie systemu, tani w realizacji.
- **Wady:** Nienaturalne dla użytkowników, trudne dla zaawansowanych użytkowników.

3. Obserwacja

- **Opis:** Testujący obserwuje użytkownika w jego naturalnym środowisku.
- **Dane:** Rzeczywiste sposoby pracy użytkowników.
- **Zalety:** Ukazuje rzeczywiste czynności użytkowników, sugeruje odpowiednie funkcje.
- **Wady:** Trudna organizacja, brak kontroli eksperymentatora.

4. Kwestionariusze

- **Opis:** Zebranie odpowiedzi na z góry określony zestaw pytań.
- **Dane:** Subiektywne opinie, elementy lubiane i nielubiane.
- **Zalety:** Łatwa powtarzalność.
- **Wady:** Konieczność pilotażu, aby usunąć niejasności.

5. Wywiady

- **Opis:** Zbieranie odpowiedzi na pytania otwarte oraz opinii w sprawach nieujętych w pytaniach.
- **Zalety:** Elastyczne, wnikliwe.
- **Wady:** Czasochłonne, trudne do analizy i porównania wyników.

6. Grupy dyskusyjne

- **Opis:** Dyskusje grupowe na temat przyszłego lub aktualnego systemu.
- **Dane:** Opinie, potrzeby i wrażenia użytkowników.
- **Zalety:** Spontaniczność reakcji.
- **Wady:** Trudność analizy wyników, niska ważność wniosków.

7. Log użytkownika

- **Opis:** Automatyczne zapisywanie działań użytkownika w systemie.
- **Dane:** Rzeczywiste sposoby korzystania z funkcji.
- **Zalety:** Możliwość ciągłego monitorowania.
- **Wady:** Wymaga analizy dużych ilości danych, może naruszać prywatność.

8. Informacje zwrotne od użytkowników

- **Opis:** Ciągłe zbieranie opinii o systemie od użytkowników.
- **Dane:** Problemy i potrzeby użytkowników.
- **Zalety:** Śledzenie zmieniających się wymagań.
- **Wady:** Konieczność utworzenia dedykowanego zespołu do obsługi feedbacku.

Podsumowanie

Funkcjonalność systemu można oceniać na różne sposoby, od obserwacji po analizę logów użytkownika. Wybór metody zależy od celu badania, liczby użytkowników i fazy cyklu życia systemu. Testy funkcjonalności są niezbędne, aby upewnić się, że system spełnia potrzeby użytkowników i działa zgodnie z oczekiwaniami.

Aspekty etyczne testowania z udziałem użytkowników

Zasady etyczne (wg Nielsen, 1993):

1. Przygotuj wszystko przed przybyciem użytkownika.
2. Podkreśl, że to system jest testowany, nie użytkownik.
3. Poinformuj użytkownika, że oprogramowanie jest nowe i może zawierać błędy.
4. Zapewnij użytkownika, że może przerwać test w dowolnym momencie.
5. Wyjaśnij używane techniki monitorujące (np. logi, nagrania).
6. Zapewnij o pełnej poufności wyników testu.
7. Odpowiedz na wszystkie pytania użytkownika.

Dalsze wskazówki etyczne:

1. Pomóż użytkownikowi osiągnąć sukces w pierwszej fazie testu.
2. Wydawaj polecenia testowe pojedynczo.
3. Utrzymuj relaksującą atmosferę (np. przerwy, kawa).
4. Unikaj zakłóceń (np. zamknięte drzwi, wyłączony telefon).

5. Nigdy nie sugeruj, że użytkownik popełnia błąd lub działa zbyt wolno.
6. Zminimalizuj liczbę obserwatorów testu.
7. Przerwij test, jeśli stanie się nieprzyjemny dla użytkownika.

Po zakończeniu testu:

1. Podkreśl, że użytkownicy pomogli w zidentyfikowaniu obszarów wymagających poprawy.
2. Nie raportuj wyników w sposób umożliwiający identyfikację użytkowników.
3. Używaj nagrań wideo tylko za zgodą uczestników.

Przykład testowania użytkowości – system FK

Cel testowania:

- Sprawdzenie, czy nowa wersja modułu systemu FK umożliwia drukowanie raportu kasowego (RK) za wybrany dzień i kasę główną.

Przygotowanie testu:

- **Obiekt testowania:** Nowa wersja systemu FK.
- **Metoda:** Testowanie użytkowe z zastosowaniem metody kierowanego głośnego myślenia.
- **Plan testowania:**
  - Uczestnicy: główna księgowa, kasjerka.
  - Sprzęt: komputer PC i drukarka.
  - Miejsce: dział księgowości firmy X.

Opis testu:

- **Zadania użytkowników:**

- Wpisywanie pozycji RK za dni 11.11.2001 i 12.11.2001.
- Wyszukiwanie RK za wybrane dni.
- Drukowanie RK za dzień 11.11.2001.
- Sprawdzenie automatycznego nadawania numerów RK.

Procedury testowe:

- **Dane wejściowe:** Dekrety z poszczególnych pozycji RK.
- **Dane wyjściowe:** RK w formie ekranu i wydruku.
- **Analiza wyników:** Sprawdzenie poprawności wprowadzonych danych na wydruku.

Wyniki testów:

- Wszystkie wymagane wyniki zostały uzyskane (wydruki, poprawne wyświetlenie RK na ekranie).
- Raport może zawierać tabelę z opisem realizacji funkcji, możliwościami realizacji oraz uwagami.

Test – Ocena użytkownika

Kryteria oceny:

1. **Międzymordzie (interfejs):**
   - Zgodność ze standardami.
   - Intuicyjność.
   - Spójność i ergonomia.
2. **Dane wejścia/wyjścia:**
   - Elastyczność.
   - Obsługa błędów.

3. **Funkcjonalność i użyteczność:**
   - Przydatność funkcji.
   - Wydajność i poprawność działania.

Podsumowanie

Testowanie z udziałem użytkowników wymaga przestrzegania zasad etycznych i skupienia się na analizie użyteczności oraz funkcjonalności systemu. Przykład testu systemu FK pokazuje, jak dobrze zaplanowane testowanie może pomóc w identyfikacji problemów i poprawie oprogramowania.

Testowanie dokumentacji i zarządzanie procesem testowym

1. Testowanie dokumentacji

Elementy testowania dokumentacji:

1. **Odbiorcy:** Czy dokumentacja odpowiada potrzebom użytkowników?
2. **Terminologia:** Czy użyte terminy są spójne i zrozumiałe?
3. **Treść i zawartość:** Czy dokumentacja zawiera pełne i poprawne informacje?
4. **Same fakty:** Czy fakty są aktualne i zgodne z rzeczywistością?
5. **Krok po kroku:** Czy procesy opisane są w sposób przejrzysty i łatwy do naśladowania?
6. **Rysunki i ekrany:** Czy wizualizacje wspierają zrozumienie tekstu?
7. **Próbki i przykłady:** Czy podane przykłady są praktyczne i adekwatne?
8. **Pisownia i gramatyka:** Czy tekst jest poprawny językowo?

2. Planowanie testów

Cel planowania testów:

- Określenie **zakresu testów.**
- Wybór **metodyki.**
- Przydzielenie **środków** i **personelu.**
- Ustalenie **harmonogramu.**
- Utworzenie szczegółowego **planu testowania.**

3. Szablony planów testowania

- **Przykładowe szablony:**

Standard ANSI/IEEE 829/1983 dla dokumentacji testowania oprogramowania.

- **Indywidualne:**

Tworzone dla każdego testu w zależności od jego charakterystyki.

4. Planowanie zadań testowych

Zadania testowe – co?

- Identyfikacja funkcji do przetestowania.
- Określenie metod testowania każdej funkcji.
- Kryteria zaliczenia i niezaliczenia.

Procedury testowe – jak?

- Instrukcje krok po kroku.
- Przygotowanie środowiska testowego.
- Odtworzenie warunków testowych.

5. Specyfikacja zadań testowych (wg IEEE)

- **Numeracja:** Unikalne oznaczenie zadania.

- **Przedmiot testu:** Co jest testowane?
- **Dane wejściowe i wyjściowe:** Jakie dane są używane i jakie wyniki są oczekiwane?
- **Wymagania środowiska:** Sprzęt, oprogramowanie, inne zasoby.
- **Ograniczenia proceduralne:** Jakie są limity lub warunki testu?
- **Zależności:** Czy test zależy od innych zadań?

6. Procedury testowe (skrypt testowy)

Etapy procedury:

1. **Cel testu:** Co chcemy osiągnąć?
2. **Specjalne wymagania:** Potrzebne zasoby.
3. **Przebieg:**
   - Przygotowanie.
   - Rozpoczęcie testu.
   - Procedura testowa.
   - Pomiar wyników.
   - Zakończenie.
   - Odtworzenie warunków.
4. **Nieprzewidziane wypadki:** Co zrobić w przypadku awarii?

7. Organizacja i zarządzanie zadaniami testowymi

Kluczowe pytania:

- Jakie zadania należy wykonać?
- Ile czasu zajmą testy?
- Jakie zadania zostały zaliczone, a jakie nie?
- Jaki procent zadań zaliczono w poprzednich przebiegach?

Formy organizacji:

1. **W głowie:** Dla prostych projektów.
2. **Papierowa:** Lista kontrolna.
3. **Arkusz kalkulacyjny:** Strukturalne podejście.
4. **Baza danych:** Dla dużych projektów.

8. Testowanie – raportowanie wyników

Zasady skutecznego raportowania błędów:

1. Raportuj jak najszybciej.
2. Utrzymaj obiektywność – nikogo nie oskarżaj.
3. Śledź losy raportu i jego naprawy.

Elementy raportu błędu:

- **Unikalne oznaczenie błędu.**
- **Opis błędu:**
  - Kiedy wystąpił.
  - Kto go wykrył.
  - Na jakim sprzęcie/systemie.
  - Dane wejściowe i procedura testowa.
  - Wyniki: oczekiwane i uzyskane.

9. Naprawa błędów

Proces naprawy:

1. Tester znajduje błąd.
2. Programista naprawia błąd.

3. Tester potwierdza naprawę.

Ważność i priorytet błędu:

- **Ważność:** Wpływ błędu na system (katastrofalny, poważny, lekki).
- **Priorytet:** Jak pilnie należy go naprawić (natychmiast, przed wdrożeniem, w miarę możliwości).

10. Dokumentacja testowa

Część stała:

- Plan testu (harmonogram).
- Specyfikacja struktury testu.
- Dane testowe i oczekiwane wyniki.
- Kryteria zaliczenia testów.

Część dynamiczna:

- Kronika testów.
- Raport zdarzeń.
- Podsumowanie wyników.

Zalecana struktura dokumentacji:

1. Jednoznaczny identyfikator.
2. Powiązania z innymi dokumentacjami.
3. Kryteria zaliczenia testu.
4. Opis środowiska testowego.
5. Rozdysponowanie zadań.
6. Harmonogram testów.

Podsumowanie

Planowanie testów i ich raportowanie to kluczowe elementy zarządzania jakością oprogramowania. Odpowiednia organizacja zadań, jasno określone procedury oraz precyzyjna dokumentacja pomagają skutecznie identyfikować i naprawiać błędy, zapewniając niezawodność systemu.

Automatyzacja testów

Podstawowe informacje o automatyzacji testów funkcjonalnych:

- Automatyzacja testów funkcjonalnych nie skraca czasu testowania ani nie ułatwia pracy w takim stopniu, jak często się uważa.
- Jest oparta na zautomatyzowaniu istniejących procedur testów manualnych.
- Wymaga solidnych podstaw w cyklu testowania oraz dobrze zaprojektowanego procesu zarządzania automatyzacją, szczególnie w przypadku testów regresyjnych.

Wymagania i inwestycje związane z automatyzacją:

1. **Czas:**
   - Wybór odpowiednich narzędzi.
   - Zapoznanie się z ich działaniem.
   - Przygotowanie skryptów testowych.
   - Interpretacja wyników testów.
2. **Pieniądze:**
   - Zakup narzędzi.
   - Koszty utrzymania automatyzacji.

Korzyści z automatyzacji:

- **Zwiększenie jakości oprogramowania:** Możliwość wykonania większej liczby przypadków testowych i eliminacja większej liczby błędów.
- **Zmniejszenie ryzyka:** Niższe ryzyko awarii po wdrożeniu na produkcję.
- **Obniżenie kosztów usuwania błędów produkcyjnych.**

Ograniczenia automatyzacji testów:

- **Nie zmniejsza czasu potrzebnego na testy.**
- **Nie zmniejsza zasobów potrzebnych do testowania.**
- **Nie wykrywa błędów niewykrywalnych w testach manualnych.**

Rodzaje narzędzi do automatyzacji testów:

1. Narzędzie wychwytywania/odtwarzania:

- Zapamiętuje pozycje myszy, kliknięcia i wejścia klawiatury.
- Umożliwia powtarzanie tych działań w przyszłości.
- **Korzyści:**
  - Przydatne w testach interfejsów GUI.
  - Umożliwia kontrolę testów GUI bez powtarzania manualnych działań.
- **Ograniczenie:** Powtarzane są jedynie wcześniej zarejestrowane działania.

2. Narzędzie do pisania skryptów testowych:

- Umożliwia tworzenie skryptów testowych w języku testowym.
- **Działanie:**
  - Skrypty definiują dane wejściowe, oczekiwane wyniki i sposób uruchamiania programu.

- Narzędzie przetwarza skrypty na wykonalne testy i generuje raporty wyników.
- **Zalety:**
  - Możliwość pełnej automatyzacji po przygotowaniu skryptów.

Przykłady narzędzi do automatyzacji testów funkcjonalnych:

1. Apache JMeter

- **Zastosowanie:** Testy wydajności, obciążeniowe, analiza danych.
- **Zalety:**
  - Open source.
  - Elastyczne i zaawansowane skrypty.
  - Graficzna prezentacja wyników.
- **Wady:**
  - Skomplikowany interfejs użytkownika.
  - Brak współpracy z Power Builderem.

2. AutoMate

- **Zastosowanie:** Automatyzacja procesów biznesowych i testów aplikacji.
- **Zalety:**
  - Intuicyjny interfejs.
  - Współpraca z różnymi aplikacjami, w tym Power Builderem.
- **Wady:**
  - Niestabilność systemu.
  - Problemy z odtwarzaniem skryptów.

3. Bad Boy

- **Zastosowanie:** Testowanie przeglądarek internetowych (Internet Explorer).
- **Zalety:**
  - Open source.
  - Przyjazny interfejs.
  - Zaawansowane funkcje, np. pętle, scheduler, asercje.
- **Wady:**
  - Brak wsparcia dla Power Buildera.

4. Vermont

- **Zastosowanie:** Testy regresyjne.
- **Zalety:**
  - Współpraca z Power Builderem.
  - Szczegółowe raporty wykonania testów.
- **Wady:**
  - Niestabilność.
  - Problemy z odtwarzaniem skryptów.
  - Brak porównywania obiektów graficznych.

5. TestComplete

- **Zastosowanie:** Testy jednostkowe, funkcjonalne, regresyjne, aplikacji rozproszonych.
- **Zalety:**
  - Stabilność systemu.
  - Możliwość tworzenia zaawansowanych skryptów.
  - Szczegółowe raporty wyników.
- **Wady:**
  - Wysoka cena licencji.

- Mało intuicyjny interfejs.
- Działa tylko na Windows.

6. QuickTest Professional

- **Zastosowanie:** Testy funkcjonalne i regresyjne.
- **Zalety:**
  - Stabilność systemu.
  - Klarowne raporty wyników.
- **Wady:**
  - Działa tylko na Windows.
  - Współpraca tylko z Internet Explorer.
  - Wysoka cena licencji.

7. Automation Anywhere

- **Zastosowanie:** Automatyzacja testów i zadań biznesowych.
- **Zalety:**
  - Intuicyjny interfejs.
  - Nie wymaga umiejętności programistycznych.
  - Stabilność systemu.
- **Wady:**
  - Brak szczegółowego raportowania.
  - Działa tylko na Windows.

Podsumowanie

Automatyzacja testów to potężne narzędzie wspomagające proces weryfikacji oprogramowania, jednak wymaga znacznych inwestycji czasu i zasobów. Przy jej wdrożeniu kluczowe jest dobranie odpowiednich narzędzi i przemyślane

zaplanowanie procesu, aby uzyskać długoterminowe korzyści w postaci poprawy jakości oprogramowania i zmniejszenia ryzyka błędów.

Podsumowanie testowania – 13 Platynowych Idei na Testowanie wg Raya Jutkinsa

Wprowadzenie

Testowanie jest kluczowym elementem procesu tworzenia oprogramowania, który pozwala na identyfikację i eliminację błędów przed wdrożeniem produktu na rynek. Ray Jutkins przedstawił 13 platynowych idei, które mają na celu zoptymalizowanie procesu testowania, zwiększenie efektywności oraz zapewnienie najwyższej jakości produktu końcowego.

1. Testowanie jako Koncepcja

1.1. Testowanie nie należy do nauk ścisłych

- **Opis:** Testowanie nie musi opierać się na rygorystycznych metodach naukowych.
- **Implikacja:** Możliwość zastosowania elastycznych i praktycznych metod testowania.

1.2. Dlaczego testować?

- **Cel:** Poznanie rynku i zrozumienie potrzeb klientów.
- **Strategia:** Przed rozpoczęciem testów należy opracować strategię i plan działania.

1.3. Elementy planu testowania

- **Plan testowania obejmuje:**

1. Koncepcję tego, czego trzeba się dowiedzieć.
2. Zaplanowanie, organizację i przygotowanie testów.
3. Unikanie zgadywania – planowanie testów na podstawie jasno określonych celów.

2. Badania i Testowanie to Nie To Samo

2.1. Czym są badania?

- **Definicja:** Badania ustalają motywacje i zamiary klientów dotyczące zakupów.
- **Techniki badawcze:**
  1. Wywiady
  2. Ankiety
  3. Grupy skupione
- **Formy prowadzenia badań:** Twarzą w twarz, telefonicznie, listownie, faksem, e-mailem.

2.2. Czym jest testowanie?

- **Definicja:** Testowanie ocenia rzeczywiste zachowanie klientów, a nie tylko ich deklaracje.
- **Cel:** Dowiedzenie się, jak klienci faktycznie działają na rynku.
- **Metoda:** Testowanie na żywo, zaufanie klientom i rynkowi.

3. Dlaczego Testować?

3.1. Powody do testowania:

1. **Nierealny czas i koszty:** Testowanie pomaga kontrolować przedłużające się terminy i rosnące koszty.

2. **Zmienność wyników:** Każdy test może przynieść inne rezultaty, co wymaga ciągłego dostosowywania.

3. **Prawo Murphy'ego:** Testowanie pomaga zwalczać przeciwności i zwiększa szanse na sukces.

3.2. Osiem rzeczy, które testowanie pomoże osiągnąć:

1. **Projektowanie kampanii:** Kompleksowe zaplanowanie kampanii marketingowej.
2. **Wybór treści:** Selekcja najlepszych tekstów reklamowych i pomysłów graficznych.
3. **Targetowanie klientów:** Skupienie się na klientach z największym prawdopodobieństwem zakupu.
4. **Segmentacja rynku:** Identyfikacja najbardziej obiecujących segmentów rynku.
5. **Optymalizacja ofert:** Wybór najlepszych ofert marketingowych.
6. **Timing:** Wybór najlepszego momentu na ofertę w zależności od pory roku.
7. **Ekspansja:** Decyzja o rozszerzeniu zasięgu działania.
8. **Ryzyko:** Unikanie niepotrzebnego ryzyka w kampanii marketingowej.

4. Kiedy NIE Testować?

4.1. Ograniczenia testowania:

- **Mała liczba klientów:** Niewystarczająca liczba respondentów do przeprowadzenia znaczącego testu.
- **Specjalne okazje:** Testowanie w czasie, gdy naturalny czas może zakłócić wyniki (np. Walentynki, Dzień Matki).
- **Niewłaściwe elementy:** Testowanie elementów, które nie przynoszą wartości (np. standardowa koperta bez unikalnych cech).

4.2. Dodatkowe powody:

1. **Brak znaczącej liczby wyników:** Testowanie bez wystarczającej ilości danych.
2. **Niejasne wyniki:** Wyniki nie dostarczają istotnych informacji.
3. **Subiektywne opinie:** Testowanie oparte na uprzednich opiniach zamiast na obiektywnych danych.
4. **Brak obiektywności:** Testowanie z nieobiektywnym nastawieniem, np. poprzez sondaże w stylu "teściowej".

5. Jak Testować?

5.1. Preferowany schemat:

- **Pirymida lub schemat blokowy:** Określenie, czego trzeba się dowiedzieć i jak to osiągnąć.
- **Kodowanie:** Każdy element kampanii marketingowej powinien być zakodowany w celu śledzenia wyników.

5.2. Rzeczy do testowania:

1. **Jedna rzecz na raz lub wszystko jednocześnie.**
2. **Wielki zysk:** Skupienie się na testowaniu elementów, które mogą przynieść największy przełom.

5.3. Przykłady rzeczy do testowania:

- Tekst reklamowy: styl, podejście, długość.
- Grafika: format, układ, wzór, styl, typ, kolory.
- Rozmiar, papier, gramatura, faktura, jakość.
- Adresaci: bazy danych, segmenty, aktywność.
- Oferty: reklamy specjalne, premie, ograniczony czas.
- Produkt: pakowanie, nazwa, powiązanie z innymi usługami.
- Gwarancja i rękojmia: pełna czy ograniczona.

- Broszurki i literatura: list zachwalający, tekst techniczny.
- Ulotki: świadectwa, historie przypadków, pytania i odpowiedzi.
- Media: direct mail, transmisja, druk, media elektroniczne.
- Narzędzia telekomunikacyjne i mieszanki dowolnych elementów.

6. Kiedy Testować?

6.1. Zawsze testuj!

- **Zasada:** Testowanie powinno być integralną częścią każdego etapu kampanii marketingowej.
- **Elastyczność:** Dostosowanie terminów testowania do specyfiki branży i rynku.

6.2. Testuj wcześniej i często:

- **Wczesne testowanie:** Pozwala na wczesne wykrycie problemów.
- **Częste testowanie:** Zapewnia ciągłe doskonalenie kampanii na podstawie bieżących danych.

7. Co Testować?

7.1. Kluczowe aspekty:

1. **Testuj jedną rzecz na raz lub wszystko jednocześnie.**
2. **Szukaj dużego zysku:** Koncentruj się na elementach, które mogą przynieść największe korzyści.

7.2. Przykładowe elementy do testowania:

1. Tekst reklamowy: styl, podejście, długość.
2. Grafika: format, układ, wzór, styl, typ, kolory.

3. Rozmiar, papier, gramatura, faktura, jakość.
4. Adresaci: bazy danych, segmenty, aktywność.
5. Oferty: reklamy specjalne, premie, ograniczony czas.
6. Produkt: pakowanie, nazwa, powiązanie z innymi usługami.
7. Gwarancja i rękojmia: pełna czy ograniczona.
8. Broszurki i literatura: list zachwalający, tekst techniczny.
9. Ulotki: świadectwa, historie przypadków, pytania i odpowiedzi.
10. Media: direct mail, transmisja, druk, media elektroniczne.
11. Narzędzia telekomunikacyjne i mieszanki dowolnych elementów.

8. Co Mierzyć?

8.1. Znaczące rzeczy:

- **Reakcje klientów:** Łączne i według różnych mediów.
- **Sprzedaż:** Łączna i według mediów.
- **Koszty:** Na użyteczną informację i sprzedaż.
- **Produkty/usługi:** Jakie produkty/usługi są kupowane.
- **Średnie zamówienie:** Średnia wartość transakcji.
- **Sposoby płatności:** Czek, karta kredytowa, przekaz pocztowy, gotówka.
- **Nowi klienci:** Ilość nowych klientów pozyskanych.

8.2. Dlaczego mierzyć?

- **Sprzedaż:** Bez sprzedaży nic się nie dzieje.
- **Decydująca sprawa:** Czy sprzedaliśmy? Czy zarobiliśmy?
- **Podejmowanie decyzji:** Testowanie dostarcza informacji potrzebnych do podejmowania inteligentnych i zyskownych decyzji.

9. Jak Mierzyć?

9.1. Formuły mierzenia:

- **Proste formuły:** Można je zastosować ręcznie (np. kalkulator i papier).
- **Zaawansowane formuły:** Wymagają użycia komputera i specjalistycznego oprogramowania.

9.2. Cel mierzenia:

- **Dowiedzenie się, co stało się na rynku.**
- **Skupienie się na liczbach i pieniądzach.**
- **Określenie celów:** Co próbujesz się dowiedzieć i jak to mierzyć.

10. Czego NIE Mierzyć?

10.1. Rzeczy bez znaczenia:

- **Dzień tygodnia dostawy poczty:** Brak wpływu na wyniki.
- **Odcienie kolorów:** Mało istotne elementy, które nie wpływają na efektywność kampanii.
- **Przechył znaczka pocztowego:** Nie ma wpływu na skuteczność direct mail.

10.2. Wyjątki:

- **Znaczące elementy:** Dzień publikacji reklamy, zmiany oferty na stronie WWW.
- **Cena:** Porównanie różnych przedziałów cenowych.

11. Kiedy Test NIE Jest Testem?

11.1. Ograniczenia:

1. **Niewystarczająca liczba wyników:** Brak danych do wyciągnięcia wniosków.
2. **Brak znaczących wyników:** Wyniki nie dostarczają istotnych informacji.
3. **Subiektywne opinie:** Testowanie oparte na uprzednich opiniach zamiast obiektywnych danych.
4. **Nieobiektywne nastawienie:** Testowanie z określonym celem zamiast odkrywania rzeczywistego zachowania rynku.

12. Kiedy Test NIE Jest Testem?

12.1. Dodatkowe ograniczenia:

1. **Brak wystarczającej liczby uczestników.**
2. **Niewłaściwe elementy testowania:** Testowanie elementów, które nie przynoszą wartości.
3. **Brak obiektywności:** Testowanie z nieobiektywnym nastawieniem.
4. **Brak sensownych wyników:** Wyniki nie dostarczają istotnych informacji.

13. Wnioski po Fakcie SĄ Nauczą Ścisłą

13.1. Korzyści z wniosków po testach:

- **Co działa:** Identyfikacja skutecznych elementów kampanii.
- **Co nie działa:** Unikanie powtarzania błędów.
- **Dlaczego:** Zrozumienie przyczyn zachowań klientów.

13.2. Co zrobić z wiedzą z testów:

- **Myślenie i planowanie:** Wykorzystanie zdobytych informacji do przyszłych działań.

- **Organizacja:** Zaplanowanie kolejnych kroków na podstawie wyników testów.

Przykład Testowania – Simple Application

Przykład Testowania 1/2

- **Data:** 10.03.2003
- **Program:** Simple Application
- **Środowiska testowe:**
    1. **Windows 2000 z Java 1.3 (Sun):**
        - Komputer z jednostką centralną Intel Celeron 300MHz.
        - Pamięć operacyjna: 192MB.
        - Karta muzyczna: Sound Blaster 16.
    2. **Windows 98 SE z Java 1.4.0 (Sun):**
        - Komputer z jednostką centralną Intel Celeron 333MHz (przetaktowany na 416MHz).
        - Pamięć operacyjna: 160MB.
        - Karta dźwiękowa: OPTi 82C929 (OPTi MAD16 compatible audio chips).
    3. **Linux (DEBIAN unstable) Kernel 2.4.19 z Java 1.4.0 (IBM):**
        - Komputer z jednostką centralną Intel Celeron 333MHz.

Przykład Testowania 2/2

- **Problemy początkowe:**
    - Problemy z kompilacją.
    - Rozwiązanie poprzez zmianę struktury katalogowej programu i inny sposób kompilacji.
- **Wyniki testów:**
    - **Windows:**
        - Problemy z wyświetlaniem ikon psów.

        - Ignorowanie kliknięć myszy w pobliżu poruszającej się pionowej kreski.
        - Źródło: Zła obsługa “słuchacza” myszy.
    - **Linux:**
        - Program nie zawsze szczekał.
        - Źródło: Niestabilna wersja Linuxa.

Podsumowanie

Testowanie jest nieodzownym elementem procesu tworzenia i wdrażania oprogramowania oraz kampanii marketingowych. Zastosowanie platynowych idei Raya Jutkinsa pozwala na:

- **Planowanie i organizację:** Przemyślane planowanie testów zwiększa ich efektywność.
- **Identyfikację i eliminację błędów:** Testowanie pozwala na wczesne wykrycie i naprawę problemów.
- **Zrozumienie rynku:** Testowanie dostarcza cennych informacji o zachowaniach i preferencjach klientów.
- **Optymalizację strategii:** Wyniki testów pomagają w podejmowaniu świadomych decyzji marketingowych.

Implementacja tych idei w praktyce prowadzi do zwiększenia jakości produktów, lepszego dopasowania do potrzeb rynku oraz efektywniejszego wykorzystania zasobów.