Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki
Katedra Systemów Cyfrowych

Praca doktorska

Testowanie funkcjonalne złożonych systemów wbudowanych

Autor: mgr inż. Rafał Kimla
Kierujący pracą: dr hab. inż. Robert Czerwiński, prof. PŚ

Gliwice, październik 2025

Niniejszą rozprawę dedykuję pamięci mojej babci Heleny, której wsparcie i wytrwałość inspirowały mnie przez cały okres studiów doktoranckich, choć nie doczekała ich ukończenia.

Podziękowania

Pragnę złożyć serdeczne podziękowania wszystkim osobom i instytucjom, które przyczyniły się do powstania niniejszej rozprawy doktorskiej.

Na ręce mojego Promotora, dr hab. inż. Roberta Czerwińskiego, prof. PŚ, składam szczególne wyrazy wdzięczności za merytoryczne wsparcie, życzliwą opiekę naukową, cierpliwość oraz inspirujące wskazówki, które towarzyszyły mi na każdym etapie badań. Szczególnie doceniam gotowość niesienia pomocy w okresach wzmożonej pracy, zwłaszcza w obliczu zbliżających się terminów. Dziękuję również Opiekunom Pomocniczym, dr inż. Robertowi Malczykowi oraz dr inż. Danucie Pamule, za cenne uwagi i konstruktywną krytykę, dzięki którym praca zyskała na klarowności i wartości poznawczej. Wyrazy wdzięczności składam Prowadzącemu seminarium doktoranckie dr hab. inż. Grzegorzowi Strozikowi, prof. PŚ, oraz Szanownym Kolegom z grupy za trafne spostrzeżenia i rekomendacje, które pomogły udoskonalić ostateczny kształt rozprawy.

Serdeczne podziękowania kieruję do Członków Zespołu, z którymi miałem przywilej współpracować i wszystkich pozostałych Osób zaangażowanych w realizację zadań projektowych. Ich profesjonalizm, otwartość na dialog oraz rzetelność w zakresie planowania, prowadzenia eksperymentów, analizy danych i wdrożeń były nieocenione.

Badania przedstawione w niniejszej rozprawie zostały zrealizowane w firmie Rockwell Automation sp. z o.o. w ramach projektu Doktorat Wdrożeniowy IV, finansowanego przez Ministerstwo Nauki i Szkolnictwa Wyższego (nr grantu DWD/4/21/2020–76/003). Dziękuję za udzielone wsparcie finansowe oraz dostęp do niezbędnych zasobów infrastrukturalnych.

Na koniec pragnę wyrazić moją najgłębszą wdzięczność Rodzinie i Bliskim — w szczególności żonie Magdalenie, mamie Bożenie, wujkowi Krzysztofowi, teściom Annie i Andrzejowi oraz przede wszystkim babci Helenie, której dedykowana jest niniejsza praca — za nieustające wsparcie, zrozumienie i cierpliwość. Wasza wiara w sens podejmowanego wysiłku była dla mnie źródłem siły i konsekwencji w dążeniu do celu.

Streszczenie

Praca doktorska poświęcona jest zagadnieniom testowania funkcjonalnego złożonych systemów wbudowanych, ze szczególnym uwzględnieniem wdrożenia nowoczesnych metod i narzędzi w środowisku przemysłowym. Obszarem badań było opracowanie i implementacja ram postępowania testowego, które pozwalają na optymalizację procesu weryfikacji systemów wbudowanych w warunkach rzeczywistych, przy uwzględnieniu ograniczeń czasowych, technicznych i organizacyjnych.

W ramach projektu przeprowadzono szczegółową analizę dostępnych strategii testowania, ze szczególnym naciskiem na automatyzację, atomizację przypadków testowych, testowanie eksploracyjne oraz podejście oparte na analizie ryzyka. Opracowano i wdrożono hybrydowy *framework* testowy oparty na języku Python, zintegrowany z narzędziami do zarządzania testami oraz środowiskiem ciągłej integracji. Pozwoliło to na znaczące zwiększenie efektywności testowania, skrócenie czasu kampanii testowej oraz poprawę jakości dostarczanych rozwiązań.

Ocenie poddano strategię testowania opartą na analizie ryzyka oraz wpływ stopnia automatyzacji testów na efektywność kampanii testowych, wykazując, że świadome zarządzanie ryzykiem oraz automatyzacja mogą znacząco skrócić czas testowania i podnieść jakość produktu. Ważnym aspektem badań była także analiza wpływu testowania opartego na doświadczeniu, a w szczególności testów eksploracyjnych, na liczbę oraz istotność wykrytych defektów. Praca pokazuje, że wykorzystanie wiedzy i intuicji inżynierów testów pozwala na wykrycie błędów, które mogłyby umknąć formalnym technikom testowym. Trzecim kluczowym celem była identyfikacja czynników organizacyjnych i technicznych sprzyjających szybkiej adaptacji nowych strategii testowania w zespołach rozproszonych, co okazało się szczególnie istotne w kontekście wdrożenia rozwiązań w międzynarodowej organizacji.

Przeprowadzone wdrożenie w przedsiębiorstwie Rockwell Automation objęło projekty, w których zastosowano autorskie rozwiązania w zakresie strategii testowej. Wyniki badań potwierdziły zasadność przyjętych rozwiązań, wykazując wzrost wykrywalności

defektów, stabilność procesu oraz możliwość elastycznego zarządzania zakresem testów w dynamicznie zmieniających się warunkach projektowych.

Praca wnosi wkład w rozwój praktyk testowania systemów wbudowanych, wskazując kierunki dalszych badań w zakresie automatyzacji, wykorzystania sztucznej inteligencji oraz standaryzacji metryk jakościowych. Opracowane ramy postępowania testowego zostały wdrożone jako standard w przedsiębiorstwie, stanowiąc fundament nowoczesnego procesu weryfikacji systemów wbudowanych.

Abstract

The doctoral dissertation is devoted to the issues of functional testing of complex embedded systems, with particular emphasis on the implementation of modern methods and tools in an industrial environment. The research area focused on the development and implementation of a testing framework that enables the optimization of the verification process for embedded systems under real-world conditions, taking into account time, technical, and organizational constraints.

As part of the project, a detailed analysis of available testing strategies was conducted, with special attention given to automation, atomization of test cases, exploratory testing, and a risk-based approach. A hybrid testing framework based on Python was developed and implemented, integrated with test management tools and a continuous integration environment. This led to a significant increase in testing efficiency, a reduction in the duration of test campaigns, and an improvement in the quality of delivered solutions.

The evaluation focused on risk-based testing strategies and the impact of the degree of test automation on the efficiency of test campaigns, demonstrating that conscious risk management and automation can significantly shorten testing time and enhance product quality. Another important aspect of the research was the analysis of the influence of experience-based testing, particularly exploratory testing, on the number and significance of detected defects. The dissertation shows that leveraging the knowledge and intuition of test engineers enables the detection of errors that might escape formal testing techniques. The third key objective was the identification of organizational and technical factors that facilitate the rapid adaptation of new testing strategies in distributed teams, which proved especially important in the context of implementing solutions in an international organization.

The implementation at Rockwell Automation covered projects in which proprietary solutions in the area of testing strategy were applied. The research results confirmed the validity of the adopted solutions, demonstrating an increase in defect detection,

process stability, and the ability to flexibly manage the scope of tests in dynamically changing project conditions.

This dissertation contributes to the development of embedded systems testing practices, indicating directions for further research in automation, the use of artificial intelligence, and the standardization of quality metrics. The developed testing framework has been implemented as a standard in the company, forming the foundation of a modern embedded systems verification process.

Spis treści

Spis rysunków

Spis tabel

Rozdział 1

Wprowadzenie

Systemy wbudowane (ang. embedded systems) można zdefiniować między innymi jako specjalistyczne, zintegrowane platformy programowo-sprzętowe dedykowanego zastosowania. W przeciwieństwie do wielofunkcyjnych komputerów osobistych, urządzenia te są zaprojektowane do wykonywania jedynie określonych zadań. Systemy wbudowane mogą być oparte na mikroprocesorach lub mikrokontrolerach i zawierają oprogramowanie przeznaczone wyłącznie dla danego układu (ang. firmware) lub system operacyjny wraz ze specjalizowanym oprogramowaniem. Bardziej złożone rozwiązania korzystają z wysokiej skali integracji specjalizowanych układów scalonych (ang. Application-Specific Integrated Circuit, ASIC) bądź systemów na czipie (ang. System-on-a-Chip, SoC).

Według prognoz, do 2030 roku wartość rynku rozwiązań wbudowanych osiągnie 161,86 miliarda dolarów, co oznacza, że rośnie on o 7,1 % rocznie. Kluczowe czynniki napędzające ten wzrost to rozwój technologii Internetu Rzeczy (ang. Internet of Things, IoT), sztucznej inteligencji oraz sieci 5G, które zwiększają zapotrzebowanie na zaawansowane systemy wbudowane [1]. Prognozuje się także, że udział systemów wbudowanych utrzymuje się obecnie na poziomie około 94 % wszystkich urządzeń elektronicznych, a liczba ta rośnie w błyskawicznym tempie wraz z rozwojem Przemysłu 4.0 i urządzeń użytku domowego. Wszystkie te systemy wymagają szczególnego podejścia do testowania, które różni się zarówno od tradycyjnych testów elektrycznych, jak i klasycznego testowania oprogramowania komputerowego, bądź też aplikacji internetowych. Rodzi to również szereg komplikacji w trakcie integracji sprzętu z oprogramowaniem i stawia przed firmami coraz to nowe wyzwania w tym skomplikowanym obszarze wiedzy.

Dotychczas stosowane metody empiryczne często stanowią wewnętrzną tajemnicę firm, są nieusystematyzowane i odbiegają od optymalnych rozwiązań. Niejednokrotnie inżynierowie i osoby zarządzające testowaniem opierają się wyłącznie na intuicji, podczas gdy nauka dostarcza szeregu możliwości i metod, które pozwalają na podejście do testowania złożonych systemów wbudowanych w sposób przemyślany i systematyczny. Warto zwrócić przy okazji uwagę na fakt, że często systemy wbudowane mają bardzo znaczący wpływ na bezpieczeństwo ludzi i pozostałych urządzeń, a co za tym idzie, nie można bagatelizować aspektów związanych z ich testowaniem, gdyż każde działanie niepożądane może w efekcie prowadzić do uszczerbku na zdrowiu, czy w najgorszym przypadku, nawet śmierci.

Celem projektu doktorskiego było przeanalizowanie metod testowania funkcjonalnego (tzw. „czarnoskrzynkowego") złożonych systemów wbudowanych, aby w efekcie zaproponować technikę lub zestaw technik pozwalający na optymalną weryfikację systemu wbudowanego o wysokim stopniu komplikacji w określonych warunkach. Aby do tego doprowadzić, niezbędne było dobranie odpowiedniej metody badawczej, przeprowadzenie długotrwałych badań na produktach, a następnie opracowanie otrzymanych wyników. Dzięki zastosowaniu zwinnych (ang. agile) metodyk wytwarzania produktów, możliwe było przeanalizowanie jakości systemu przed i po przeprowadzeniu badań poprzez obserwację metryk, takich jak efektywność przypadków testowych, pokrycie wymagań testami, udział testów zautomatyzowanych, udział testowania eksploracyjnego i innych.

Należy podkreślić, że wyczerpujące (ang. exhaustive) przetestowanie tak złożonych produktów jest niemożliwe do przeprowadzenia z przyczyn czasowych i ekonomicznych. Dlatego też kluczowym aspektem dla dalszego rozwoju systemów wbudowanych jest nie tylko dopracowanie metod ich testowania, ale także określenie optymalnych ram dla zakresu testów. Bardzo istotnym elementem jest więc automatyzowanie testowania regresyjnego oraz minimalizacja zakresu tak zwanych retestów. Projekt doktorski porusza ponadto wpływ automatyzacji testów regresyjnych i wprowadzenia środowiska ciągłej integracji, jak również filozofii *DevOps* (ang. Development and Operations), która integruje rozwój, eksploatację oraz aktywności związane z zapewnieniem jakości produktów.

Praca wdrożeniowa była realizowana w ramach działalności badawczo-rozwojowej przedsiębiorstwa Rockwell Automation w obszarze testowania przemienników częstotliwości niskiego i średniego napięcia Allen-Bradley PowerFlex, kontrolerów silnikowych

serii ArmorStart, ArmorPowerFlex, produktów powiązanych, takich jak karty rozszerzeń komunikacyjnych, enkoderowych, wejść-wyjść oraz sterowników programowalnych (ang. Programmable Logic Controller, PLC) Allen-Bradley ControlLogix i adapterów sieciowych z serii 1756-EN4. Jakość wytwarzanych produktów jest jednym z najważniejszych aspektów dla firmy Rockwell Automation, która poza inwestowaniem w budowę, certyfikację i rozwój laboratoriów testowych, ciągle poszukuje efektywnych metod testowania swoich produktów i ulepszania istniejących procesów.

Analiza efektywności metod testowania funkcjonalnego złożonych systemów wbudowanych pozwoliła przedsiębiorstwu, w którym toczyły się prace wdrożeniowe, na optymalizację środków przeznaczonych na testowanie, jak również przełożyła się na wzrost jakości dostarczanych rozwiązań. Należy zauważyć, że projektowanie oraz produkcja tak skomplikowanych urządzeń wiążą się z ogromną liczbą potencjalnych defektów, które ponadto z czasem mają tendencję do uodparniania się na istniejące testy, co w teorii testowania oprogramowania jest znane pod nazwą paradoksu pestycydów (ang. pesticide paradox). Co za tym idzie, poszukiwanie skuteczniejszych technik testowania jest niezwykle istotne dla jakości końcowego systemu.

Intencją projektu tworzonego w ramach programu „Doktorat wdrożeniowy IV" była przede wszystkim możliwość implementacji wyników przeprowadzonych badań naukowych w sektorze przemysłowym i wykazanie, że istnieje taki zestaw technik testowania, które wdrożone w formie *frameworku*[1] pozwalają na optymalizację weryfikacji złożonego systemu wbudowanego w określonych warunkach. Przeprowadzona analiza i ocena efektywności metod testowania funkcjonalnego złożonych systemów wbudowanych miała na celu znalezienie technik, które pozwolą zoptymalizować proces testowania urządzeń, a co za tym idzie usprawnić procesy, wygenerować oszczędności, czy wreszcie skrócić czas potrzebny na dostarczenie gotowego rozwiązania do klientów. Można także założyć, że zastosowanie zaproponowanych metod testowania wpływa pozytywnie na końcową jakość produktu.

[1] Autor jest świadomy istnienia polskiego odpowiednika angielskiego terminu „framework", tj. „ramy postępowania". Tłumaczenie to nie oddaje jednak w pełni natury oryginalnego terminu oraz nie jest powszechnie używane zarówno przez programistów jak i w środowisku akademickim. W przypadku stosowania tego rodzaju zapożyczeń z języka angielskiego w niniejszej pracy (np. „debugowanie" zamiast „odpluskwianie") stosowana będzie pisownia kursywą.

Rozdział 2

Analiza literatury i teoria testowania systemów wbudowanych

2.1 Charakterystyka złożonych systemów wbudowanych

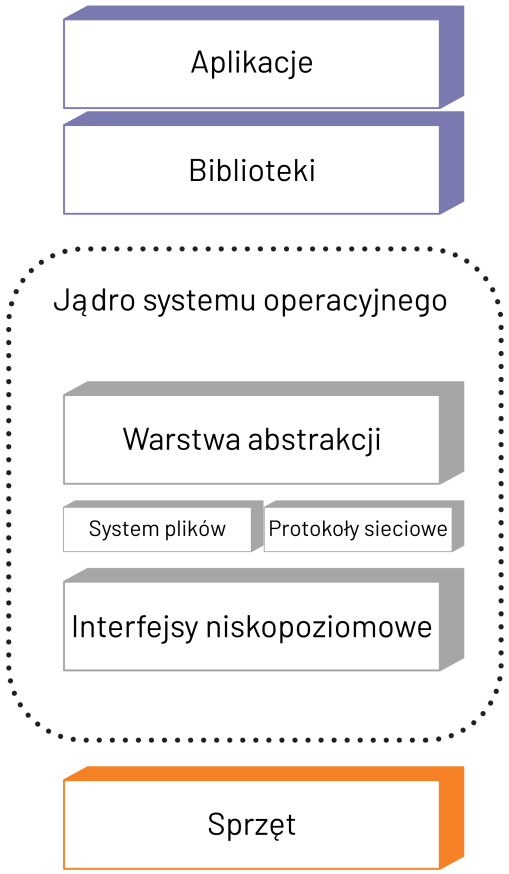
Ze względu na indywidualną architekturę każdego systemu, projektowanie systemów wbudowanych stanowi złożony proces wymagający od inżynierów szerokiego zakresu kompetencji — od architektury systemowej, przez projektowanie platform sprzętowych i obwodów drukowanych, po wykorzystanie systemów operacyjnych oraz programowanie sterowników, warstw komunikacyjnych i aplikacji systemowych.

Architektura systemów wbudowanych przypomina architekturę systemu komputerowego, dzieląc się na mikroprocesor, pamięć i urządzenia peryferyjne. Ze względu na unikatowy projekt, zarówno wytwarzanie sprzętu, jak i oprogramowania stanowi wyzwanie w porównaniu do aplikacji komputerowych opartych o znaną architekturę. Sytuację komplikują ponadto ograniczone zasoby w postaci pamięci i mocy obliczeniowej systemów wbudowanych. Złożone systemy wbudowane coraz częściej opierają więc swoje działania na dedykowanych systemach operacyjnych zarządzających zasobami systemowymi i kontrolujących działanie aplikacji [2].

Obecnie jednym z najbardziej popularnych rozwiązań, jeśli chodzi o systemy operacyjne dla systemów wbudowanych, jest stosowanie różnych dystrybucji Linuxa [3]. Wymusza to spełnienie przez taki system kilku wymogów:

- minimum 32-bitowego mikroprocesora wyposażonego w jednostkę zarządzania pamięcią (ang. Memory Management Unit, MMU),
- posiadania wystarczającej ilości pamięci operacyjnej (w zależności od danej dystrybucji Linuxa),
- zapewnienia interfejsów wejścia-wyjścia jako minimum umożliwiających *debugowanie* oprogramowania,
- jądro systemu musi być w stanie załadować system plików root za pomocą jakiejś formy trwałej pamięci lub uzyskać do niego dostęp przez sieć.

Przykładowa architektura takiego systemu na wysokim poziomie abstrakcji przedstawiona została na Rysunku 2.1.



Rysunek 2.1: Przykładowa architektura systemu wbudowanego opartego o system operacyjny Linux

Oprócz ograniczeń wynikających z zastosowanego systemu operacyjnego, istotnym wyzwaniem jest rosnąca złożoność systemów wbudowanych, wynikająca zarówno z wymagań funkcjonalnych, jak i niefunkcjonalnych. Wybór odpowiedniej platformy sprzętowej i programowej jest kluczowy, ponieważ determinuje możliwości implementacyjne oraz efektywność końcowego rozwiązania. Staje się to coraz częściej wielowymiarowym zagadnieniem, gdyż nie wystarcza dobór sprzętu spełniającego wymagania techniczne. Takie podejście sprawdzało się dobrze w przypadku prostych i nieskomplikowanych aplikacji wdrażanych na systemach z niewielką lub żadną zależnością od systemów zewnętrznych. Jednak w obecnej sytuacji złożonych systemów i systemów systemów (ang. systems of systems), gdzie istnieją skomplikowane interfejsy między nimi i większa zależność od siebie nawzajem, rozwój aplikacji wbudowanych stał się wyzwaniem. Platforma wbudowana dla takich aplikacji nie może być wybierana wyłącznie na podstawie wymagań produktu. Projektanci systemów wbudowanych muszą brać pod uwagę interfejsy systemów zewnętrznych, takie jak scenariusze wdrożeniowe, przypadki użycia produktu, trudności w utrzymaniu lub uruchomieniu, interfejsy logiczne systemu, interfejsy procesów oraz wyzwania związane z rozwojem przy wyborze platformy, a tym samym skuteczną realizację produktu [4].

Sytuacja ta powoduje, że choć projektanci systemów wbudowanych mają tendencję do używania standardowych mikroprocesorów ogólnego przeznaczenia z niewielką lub żadną specyficzną konfiguracją sprzętową, to w przypadku złożonych systemów wbudowanych coraz częściej sięga się po specjalizowane układy scalone lub systemy na czipie [5].

2.2 Wyzwania i tendencje rozwojowe

W 2002 roku już ponad 90% specjalizowanych układów scalonych zawierało mikroprocesory wykonane w technologii 130 nm [6]. Obecny rozwój technologii doprowadził do stworzenia pierwszych specjalizowanych układów stosowanych w koparkach kryptowalut w technologii 3 nm [7]. Przewiduje się, że w roku 2025 proces technologiczny osiągnie poziom 2 nm, a w 2027 roku poziom 1 nm [8]. Większa gęstość tranzystorów wynikająca ze stosowania technologii litograficznej 7 nm i poniżej, wprowadza nowe ryzyka związane z defektami procesowymi w ekstremalnie małych trójwymiarowych strukturach, defektami parametrycznymi i starzeniem się układów, co z kolei skutkuje rozwojem dziedziny projektowania pod kątem testowania (ang. Design for Testabili-

ty, DfT) [9]. Dzięki temu mechanizmy testowe stają się nieodłączną częścią projektu układu i możliwe staje się stosowanie coraz bardziej zaawansowanych technik, takich jak testy strukturalne, czy stresowe, a także monitorowanie i diagnostykę w trakcie eksploatacji.

W przypadku systemów na czipie, dzięki poprawie gęstości standardowych komórek i komórek bitowych na poziomie węzła, możliwe jest zintegrowanie większej liczby funkcji w danym obszarze systemu. Zakłada się, że obszar integracji mobilnych SoC pozostaje stały na poziomie 80 mm^2 w kolejnych generacjach. W związku z tym ilość pamięci oraz procesorów graficznych (ang. Graphics Processing Unit, GPU) podąży za skalowaniem gęstości ulotnej pamięci statycznej (ang. Static Random-Access Memory, SRAM) i standardowych komórek, zakładając, że trend na bardziej równoległe architektury będzie kontynuowany [8]. Rosnąca skala integracji wraz ze zmniejszającym się procesem technologicznym powoduje szereg problemów termicznych. Jeśli nie dojdzie do przełomu w tym zakresie, częstotliwość taktowania mikroprocesora będzie musiała być coraz częściej ograniczana, aby utrzymać tę samą gęstość mocy.

Mimo ograniczeń technologicznych i termicznych, trend budowania heterogenicznych wieloprocesorowych systemów na czipie (ang. multiprocessor-system-on-a-chip, MPSoC) dalej się nasila. Układy te składają się z wielu pracujących równolegle procesorów i rdzeni do zastosowań takich jak terminale mobilne, dekodery, procesory gier, procesory wideo i procesory sieciowe. Często zawierają bardzo zaawansowane sieci komunikacyjne zwane sieciami na chipie (ang. Network-on-a-Chip, NoC) [6]. Rosnąca presja czasowa i coraz krótsze okno rynkowe (czyli okres, w którym wprowadzenie produktu na rynek jest najbardziej korzystne), bardziej złożone funkcjonalności i rosnąca niezawodność dostarczają kolejnych wyzwań dla projektantów i sprawiają, że potrzebne są fundamentalne zmiany względem metod projektowania specjalizowanych układów scalonych, które nie są skalowalne dla wieloprocesorowych systemów na czipie. Powoduje to powstawanie kolejnych warstw abstrakcji i idące za tym skomplikowanie interfejsów sprzętowo-programowych, które z jednej strony obsługują wywołania poszczególnych funkcji lub instrukcji, a z drugiej fizyczne połączenia [10].

Wspomniane warstwy abstrakcji prowadzą do tworzenia systemów wbudowanych definiowanych programowo (ang. software-defined embedded systems), będących częściowo lub całkowicie niezależnych od warstwy sprzętowej. Dynamiczny rozwój oprogramowania, znacznie wyprzedzający rozwój warstwy sprzętowej, prowadzi coraz częściej do uruchamiania specjalistycznego dotąd oprogramowania wbudowanego na kompute-

rach ogólnego zastosowania — osobistych, klasy przemysłowej, czy nawet mikrokomputerach. Jednym z przykładów takiego zastosowania może być automatyka definiowana programowo (ang. Software-Defined Automation, SDA) [11]. Podejście to pozwala na natywne uruchomienie kodu sterownika przemysłowego lub innego urządzenia automatyki na komputerze przemysłowym z wykorzystaniem systemu operacyjnego czasu rzeczywistego (ang. Real-Time Operating System, RTOS). W zależności od mocy platformy sprzętowej możliwe jest dynamiczne alokowanie zasobów — na przykład rdzeni procesora lub pamięci — do potrzeb użytkownika.

Nie bez znaczenia pozostają też wyzwania i tendencje rozwojowe związane z modelem wytwarzania systemów wbudowanych. Ze względu na czynniki takie jak presja czasu, produktywność, innowacyjność i satysfakcja pracowników, a także rosnący poziom skomplikowania oprogramowania, w przemyśle stosuje się szeroką gamę metodyk zwinnych (ang. agile) [12]. Według statystyk, w sektorze wytwarzania oprogramowania obecnie aż 86% firm wykorzystuje Scrum[1] lub inną metodykę do codziennej pracy i przewiduje się, że liczba ta będzie rosnąć [13]. Podejście polegające na inkrementalnym wzroście w zakresie nowych funkcjonalności powoduje, że coraz częściej systemy wbudowane debiutują na rynku, posiadając zbiór podstawowych funkcjonalności, a ich rozszerzenia dostarczane są w formie aktualizacji oprogramowania. Efektem tego jest coraz bardziej istotna rola zadań związanych z testowaniem i utrzymywalnością takich systemów.

Stosowanie zwinnych metodyk wytwarzania oprogramowania niesie za sobą dodatkowe wyzwania jakościowe. Praktyki takie jak programowanie ekstremalne (ang. Extreme Programming, XP), zyskują coraz większą popularność także w zakresie rozwijania systemów wbudowanych, oferując bezpośrednią komunikację i minimalną dokumentację, co jest atrakcyjne dla programistów. Część z tych ram postępowania jest jednak trudnych do pełnego wdrożenia, co w efekcie powoduje rezygnację z części ich elementów (takich jak np. stała obecność przedstawiciela klienta) i wprowadzenie ryzyk jakościowych dla produktu [14].

[1]Iteracyjny i przyrostowy *framework* w zarządzaniu projektem, stosowany w realizacji przedsięwzięć w oparciu o metodyki zwinne wytwarzania oprogramowania.

2.3 Rola testowania w systemach wbudowanych

Testowanie jest procesem obejmującym planowanie, przygotowanie oraz ewaluację oprogramowania i powiązanych produktów, mającym na celu weryfikację zgodności z wymaganiami, potwierdzenie realizacji założonych celów oraz identyfikację usterek [15]. Niektóre źródła jako dziedzinę testowania wskazują również przewidywanie defektów, co jest możliwe za pomocą wyspecjalizowanych narzędzi lub technik uczenia maszynowego (ang. Machine Learning, ML) [16–19]. W przypadku testowania systemów wbudowanych można mówić o szczególnej istotności tego procesu ze względu na szereg czynników:

- wysoką złożoność oprogramowania i integrację ze sprzętem elektronicznym [20], czy też urządzeniami automatyki przemysłowej,
- krytyczny wpływ systemów wbudowanych na zdrowie i bezpieczeństwo użytkowników,
- standardy przemysłowe, umowy i pozostałe akty prawne,
- wysokie koszty projektowe, a co za tym idzie możliwe straty finansowe spowodowane awarią.

Podstawowym celem testowania jest określenie ryzyka powiązanego z testowanym oprogramowaniem i dostarczenie informacji dotyczącej znalezionych defektów [21]. Proces ten pomaga w ocenie jakości produktu i budowie zaufania, a także poprawie pozostałych procesów wytwarzania i zmniejszeniu liczby generowanych defektów w przyszłości. Oprócz weryfikacji produktu względem specyfikacji wymagań, istotnym aspektem jest przeprowadzenie walidacji — sprawdzenia, czy tworzony system spełni oczekiwania użytkowników.

W przypadku systemów wbudowanych wyróżnia się szereg typów testów. W zależności od poziomu granulacji można wyróżnić testy modułowe lub jednostkowe (ang. unit test), które odpowiadają za sprawdzenie pojedynczego komponentu oprogramowania. Kolejnym etapem są testy integracyjne, w ramach których weryfikowane jest łączenie poszczególnych modułów w jedną aplikację oraz współpraca między oprogramowaniem i sprzętem. Testy funkcjonalne skupiają się na testowaniu poszczególnych funkcji systemu wbudowanego w oparciu o specyfikację wymagań funkcjonalnych, a testy syste-

mowe obejmują działanie całościowe systemu lub systemu systemów (np. urządzenia wchodzącego w skład złożonej instalacji) [21].

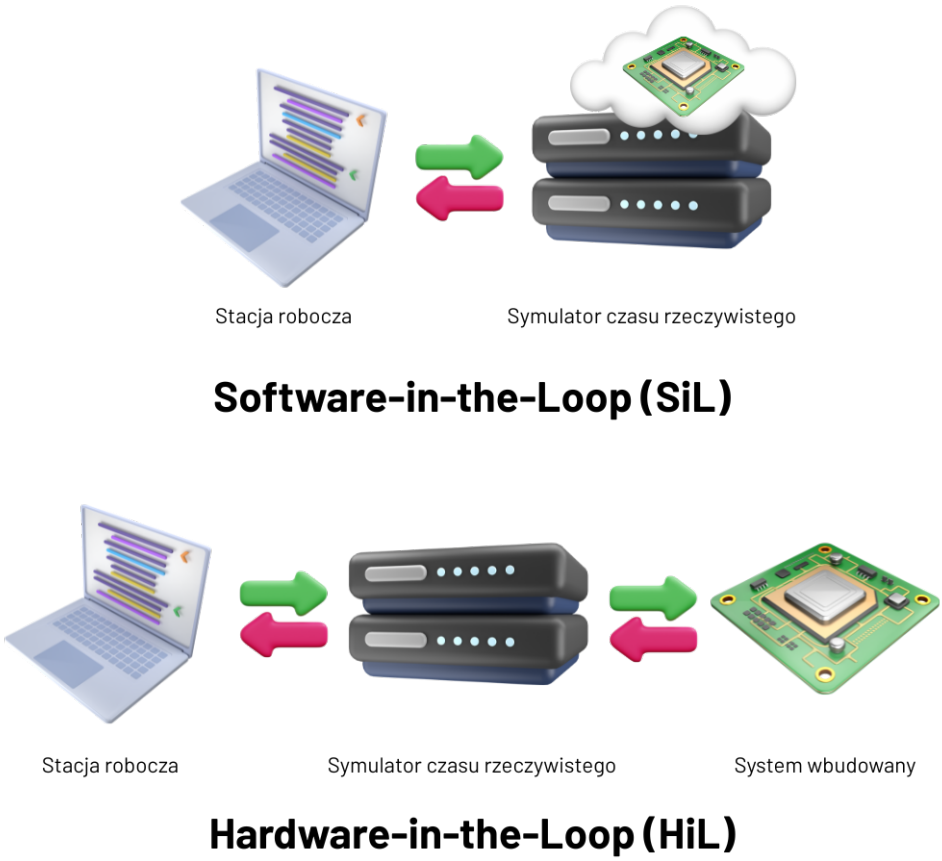
Ponadto ze względu na cykl życia oprogramowania rozróżnia się testy regresji, w ramach których po wprowadzeniu zmian do przetestowanego wcześniej obszaru oprogramowania sprawdza się poprawność i wpływ tychże zmian na działanie systemu. Kolejnymi istotnymi typami testów są tzw. retesty (testy powtórne), testy produkcyjne, testy pielęgnacyjne i testy akceptacyjne — wykonywane zgodnie z planem testów akceptacyjnych (ang. acceptance test plan) na podstawie którego klient lub użytkownik dokonuje odbioru systemu wbudowanego [21].

Testowanie systemów wbudowanych wymaga uwzględnienia zarówno komponentów sprzętowych, jak i oprogramowania, co może prowadzić do powstawania złożonych scenariuszy testowych, z kolei integracja różnych modułów sprzętowych i programowych może prowadzić do trudnych do wykrycia defektów. Bardzo często reprodukcja zgłoszonych przez testerów[2] anomalii zajmuje więcej czasu, niż sama ich naprawa. Systemy wbudowane mają ograniczoną ilość pamięci, co wymaga optymalizacji kodu i testów pod kątem efektywnego wykorzystania zasobów, a ograniczona moc obliczeniowa wymaga testowania wydajnościowego, aby upewnić się, że system działa płynnie w rzeczywistych warunkach. W systemach zasilanych bateryjnie ważne jest testowanie zużycia energii, aby zapewnić długą żywotność baterii. Systemy wbudowane często muszą działać w czasie rzeczywistym, co oznacza, że muszą spełniać określone wymagania czasowe. Ważne jest testowanie, czy system reaguje w odpowiednim czasie na zdarzenia zewnętrzne oraz testowanie w warunkach rzeczywistych, aby upewnić się, że system spełnia wymagania czasowe w różnych scenariuszach operacyjnych. Są to unikatowe wyzwania, które odróżniają testowanie systemów wbudowanych od tradycyjnych aplikacji komputerowych, sieciowych, czy mobilnych.

Rosnące oczekiwania rynkowe związane z szybszym dostarczaniem nowych rozwiązań powodują, że w przypadku testowania systemów wbudowanych coraz bardziej istotne znaczenie zyskuje środowisko testowe, które umożliwi prowadzenie testów na możliwie najwcześniejszym etapie rozwoju urządzenia. Skutkuje to stosowaniem symulatorów i emulatorów, a także systemów testowania sprzętu lub oprogramowania w pętli (ang. Hardware-in-the-Loop, HiL lub Software-in-the-Loop, SiL), których celem jest możliwie wierna imitacja rzeczywistych warunków pracy lub elementów współpracujących

[2]W niniejszej pracy termin tester stosowany jest do inżynierów testów, choć w literaturze najczęściej odnosi się do urządzenia testującego.

(np. czujników, aktuatorów, ruchu sieciowego), co jest swego rodzaju kompromisem, umożliwiającym przyspieszenie prac badawczo-rozwojowych [22]. Zasadę ich działania obrazuje schemat blokowy przedstawiony na Rysunku 2.2. Kolejnym krokiem jest użycie prototypów i testowanie na wczesnych wersjach sprzętu, aby wykryć problemy na wczesnym etapie. Zazwyczaj dopiero końcowe testy regresyjne wykonywane są na sprzęcie produkcyjnym lub bliskim produkcyjnemu.



Rysunek 2.2: Architektura systemu HiL oraz SiL

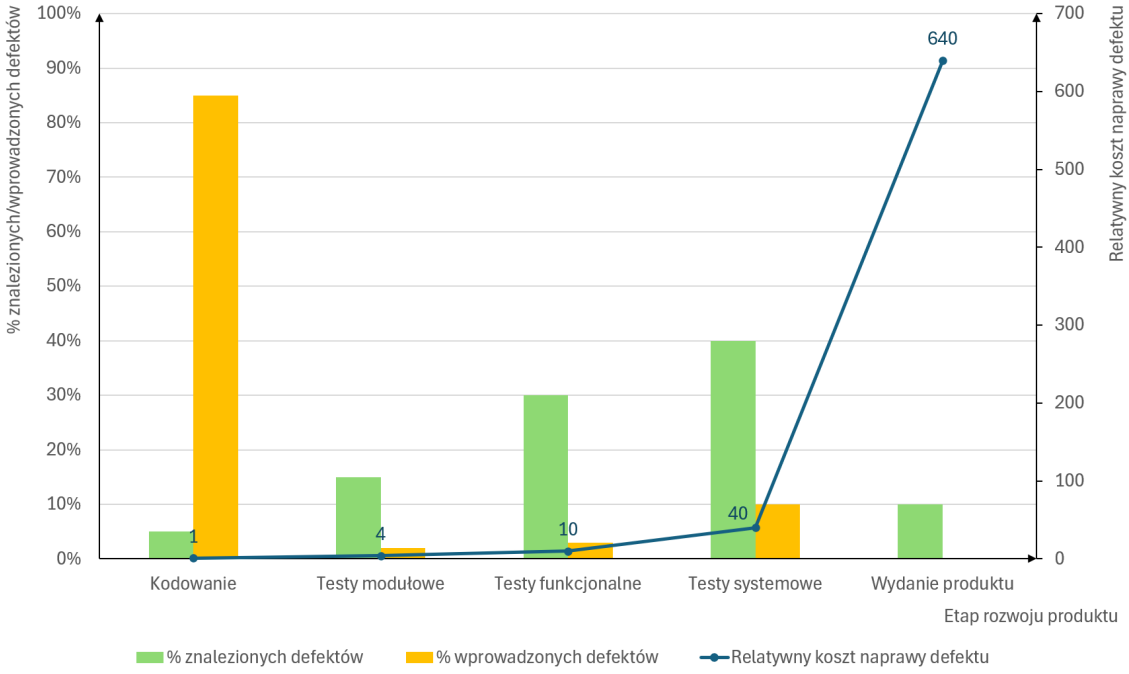
Sam proces testowania i dokumentacji nie różni się zbytnio od podejścia znanego z klasycznego testowania oprogramowania lub sprzętu [23], przedstawionego na Rysunku 2.3. Planowanie testów rozpoczyna się od zdefiniowania, co dokładnie ma być przetestowane i jakie są oczekiwane rezultaty. Ustalane są terminy dla poszczególnych etapów testowania, identyfikowane są potrzebne zasoby oraz narzędzia. Kolejnym etapem jest opracowanie scenariuszy testowych i przygotowanie danych wejściowych (wymagań, przypadków użycia, standardów przemysłowych, itd.), a także kryteriów

akceptacji — warunków, które muszą być spełnione, aby test został uznany za zaliczony. Po przeprowadzeniu testów następuje faza analizy wyników i raportowania [24]. Ze względu na przyjęty *framework*, czynności te mogą następować jedna po drugiej lub występować w różnych proporcjach w trakcie cyklu rozwoju oprogramowania, przyrastając inkrementalnie.

Rysunek 2.3: Przebieg procesu testowego

Testowanie już na etapie projektowania pozwala na wczesne wykrycie potencjalnych problemów, co może znacznie obniżyć koszty ich naprawy, jak obrazuje Rysunek 2.4. Czynności testerskie mogą odbywać się jeszcze przed powstaniem działającego prototypu urządzenia poprzez przegląd wymagań funkcjonalnych i niefunkcjonalnych, aby upewnić się, że są one kompletne, spójne i wykonalne. Sam symulator, bądź prototyp systemu wbudowanego również podlega testowaniu względem specyfikacji. Wsparcie systemu coraz częściej wkracza także w fazę utrzymania i obejmuje testowanie poprawek i aktualizacji, a także monitorowanie działania systemu w rzeczywistych warun-

kach.

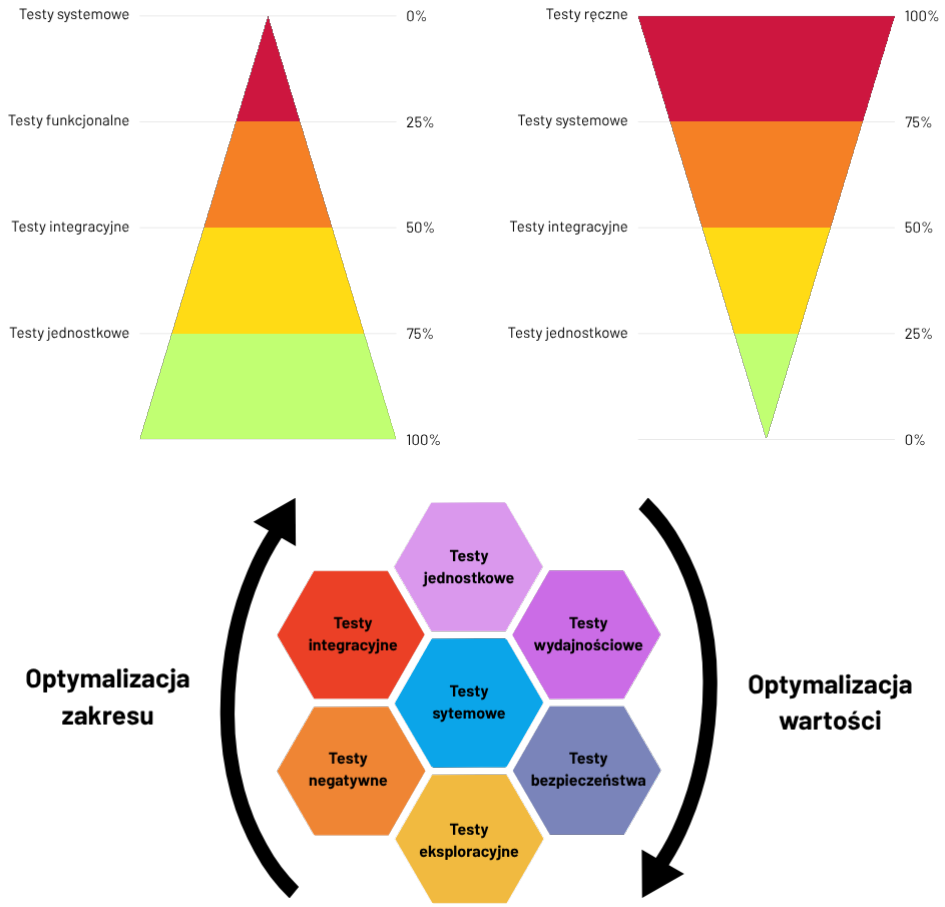


Rysunek 2.4: Wpływ wcześniejszego testowania na koszty naprawy defektów

Planowanie testów jest jednym z najważniejszych elementów całego procesu. Plan przyjmuje postać dokumentu, który opisuje cele testowania, zakres, strategię, środowisko, dane wejściowe, raportowanie, automatyzację, zarządzanie ryzykiem, kryteria wejścia i wyjścia z procesu oraz definiuje interesariuszy. Szczególne znaczenie ma dobór właściwej strategii — analitycznej (np. w oparciu o specyfikację wymagań [25], analizę ryzyka, standardy przemysłowe lub inną dokumentację), opartej na modelu systemu, reaktywnej (zależnej od jakości dostarczonego oprogramowania), czy bazującej na wiedzy eksperckiej i doświadczeniu — bądź strategii mieszanej (ang. blended test strategy), zawierającej elementy kilku różnych strategii.

Strategia testowania powinna także określać udział poszczególnych typów testów w całym procesie testowym. Najprostszym modelem jest tzw. *piramida testów*, która wskazuje, że wraz ze spadkiem poziomu granulacji, testowanie powinno zajmować mniej czasu [26]. Opiera się ona na założeniu, że szczegółowe pokrycie komponentów systemu automatycznymi testami jednostkowymi jest najbardziej efektywne. Piramida testów jest często krytykowana ze względu na pojawienie się wielu zależności i możliwych defektów na etapie integracji poszczególnych modułów systemu, w użytkowaniu poszczególnych funkcjonalności oraz interakcji z innymi systemami. Proponuje się za-

tem alternatywne podejścia, widoczne na Rysunku 2.5, takie jak odwrócona piramida (która podkreśla istotność testów ręcznych, systemowych i funkcjonalnych) czy plaster miodu (wskazujący na różnice pomiędzy wartością poszczególnych typów testów a możliwym do uzyskania pokryciem testowym).



Rysunek 2.5: Piramida testów oraz podejścia alternatywne

Mnogość zastosowań systemów wbudowanych sprawia, że rola testowania jest szczególnie istotna i wymagająca. W przypadku przemysłu motoryzacyjnego można spotkać się z systemami wspomagania kierowcy, *infotainment*, czy sterownikami silnika. Systemy wbudowane obecne w medycynie to między innymi urządzenia diagnostyczne, implanty medyczne, systemy monitorowania pacjentów. W przemyśle będą to urządzenia automatyki, systemy wizualizacji, czujniki i systemy pomiarowe. Dla użytkowników domowych będą to z kolei urządzenia internetu rzeczy: automatyka domowa, urządzenia AGD, czy telewizory i systemy audio. Z uwagi na różnorodność systemów

wbudowanych, każdy z nich wymaga innych środków w podejściu do testowania, aby uwzględnić specyficzne problemy danego systemu. Chociaż istnieje wiele powodów, dla których różne systemy wbudowane muszą być testowane w zupełnie inny sposób, istnieje również wiele wspólnych problemów, które mają podobne rozwiązania i wpisują się w każde podejście do testowania. Pewne podstawowe zasady testowania muszą mieć zastosowanie do wszystkich projektów testowania systemów wbudowanych [27]. Celem projektu i pracy doktorskiej było opracowanie tych metod i przedstawienie w formie *frameworku*.

2.4 Metody testowania funkcjonalnego

Testowanie funkcjonalne to etap testowania, w którym system weryfikowany jest względem zgodności ze specyfikacją wymagań funkcjonalnych, opisujących komponenty podlegające implementacji. Korzysta z tak zwanych *technik czarnoskrzynkowych* (ang. black-box testing) lub *szaroskrzynkowych* (ang. grey-box testing), które zakładają brak dostępu, bądź wyłącznie częściowy dostęp do kodu oprogramowania lub schematów sprzętowych. Testerzy przyjmują więc role bliskie użytkownikom końcowym i często dokonują również walidacji systemu — określenia, czy spełnia on oczekiwania klientów.

Modelowanie wymagań funkcjonalnych systemu często przedstawiane jest w formie symbolicznej w trójwymiarowej przestrzeni, gdzie każda z osi reprezentuje inny aspekt: dane, procesy oraz kontrolę [28]. Techniki testowania skoncentrowane na danych sprawdzają, czy system poprawnie przetwarza informacje z określonych obszarów tematycznych. Testy oparte na zachowaniu skupiają się na analizie dynamicznych reakcji systemu, które zależą od jego aktualnego stanu, z kolei testy regułowe badają, czy system przestrzega ustalonych zasad działania, niezależnie od jego stanu — są to zazwyczaj reguły ogólne, takie jak zasady biznesowe.

Techniki testowania można podzielić również na statyczne — bez uruchomienia systemu, a także dynamiczne — w trakcie działania urządzenia. Do technik statycznych zaliczają się przede wszystkim przeglądy (nieformalne oraz formalne) oraz inspekcje. Podstawowe techniki dynamiczne to:

- klasy (przedziały) równoważności,
- analiza wartości brzegowych,

- tablice decyzyjne,
- grafy przejść,
- testowanie losowe oraz rozmyte (ang. fuzz testing),
- testowanie metamorficzne,
- testowanie w oparciu o przypadki użycia (ang. use cases).

Testowanie na podstawie klas równoważności polega na dokonaniu podziału wektorów wejściowych na grupy pozwalające otrzymać na wyjściu identyczną odpowiedź systemu. Wyzwanie dla testera polega na zaprojektowaniu minimalnego zestawu testów, tak by każda klasa reprezentowała zestaw danych testowych o podobnych cechach i specyfikacjach. Z każdego zbioru wybierany jest jeden wektor wejściowy (przypadek), którego przetestowanie jest tożsame z weryfikacją całej klasy. Dzieje się tak, ponieważ zakłada się, że wszystkie wektory w danej klasie będą traktowane przez oprogramowanie i sprzęt w ten sam sposób. Jeśli jeden wektor w równoważnym przedziale działa, zakłada się, że wszystkie inne też będą działać, a jeśli jeden z wektorów nie działa, zakłada się, że żaden z nich nie będzie działać [29].

Jedną z najczęściej popełnianych przez programistów pomyłek jest niepoprawne wpisywanie wartości granicznych w instrukcjach warunkowych. Analiza wartości brzegowych opiera się na testowaniu na granicach między różnymi klasami lub przedziałami wektorów wejściowych. Weryfikowane są zarówno poprawne, jak i błędne wartości graniczne — liczba defektów występujących na granicach przedziałów jest stosunkowo większa niż w „środku". Jeśli wektor wejściowy określa zakres wartości między m a n, przypadki testowe projektuje się z wartościami m i n oraz tuż powyżej i tuż poniżej tych wartości, tj. $\{m-1, m, n, n+1\}$ [29].

Jeśli wektor wejściowy określa liczbę wartości, test powinien obejmować najniższe i najwyższe z nich oraz sąsiednie według powyższego schematu. Tak samo postępuje się w przypadku struktur danych o określonych wartościach brzegowych (np. tablice). W projektowaniu testów funkcjonalnych stosuje się także kombinację metody testowania na podstawie klasy równoważności z metodą analizy wartości brzegowych, co określa się terminem testowania domenowego [30–32]. W takiej sytuacji przypadek testowy obejmuje wartości graniczne, sąsiednie z granicznymi oraz wybraną dodatkową wartość z danego przedziału równoważności.

Testowanie oparte na tablicach decyzyjnych zakłada, że każdy program można uznać za funkcję, która mapuje wartości wektorów wejściowych na wartości odpowiedzi systemu. Technika ta jest także ściśle związana, a w pewnym sensie wyewoluowała, z innych *technik czarnoskrzynkowych*, takich jak testowanie na podstawie klas równoważności i testowanie wartości granicznych. Tablice decyzyjne opisują w sposób czytelny dla człowieka specyfikację funkcjonalną systemu, co jest przydatne w przypadku dokumentowania złożonej logiki. Wektory wejściowe i wynikające z nich odpowiedzi systemu tworzą wiersze tablicy. Każda kolumna odpowiada regule decyzyjnej, która opisuje wektor wejściowy powodujący oczekiwaną odpowiedź, a w przypadku złożonych reguł stosuje się techniki kombinatoryczne w celu ich optymalizacji. Wartości wektorów i odpowiedzi przedstawia się w formie wartości logicznych, tj. P (Prawda), F (Fałsz), - (warunek obojętny). Wpisy obojętne zmniejszają liczbę jawnych reguł, sugerując istnienie reguł niejawnie określonych. Taka struktura gwarantuje, że rozważona zostanie każda możliwa kombinacja wartości warunków i pozwala uzyskać minimalny oraz kompletny zestaw przypadków testowych [33].

Grafy przejść tworzone są na podstawie tablicy przejść i przedstawiają poprawne przypadki przejść pomiędzy stanami systemu. Testy przejść między stanami można zaprojektować w sposób zapewniający pokrycie typowej sekwencji stanów, przetestowanie wszystkich stanów bądź przetestowanie wszystkich przejść, konkretnych sekwencji przejść lub przejść niepoprawnych [34, 35].

Testowanie losowe polega na losowym wybieraniu danych testowych z przestrzeni wejściowej badanego systemu, zgodnie z określonym rozkładem prawdopodobieństwa. Jest to szczególnie przydatne, gdy wiedza o domenie testowanego systemu jest ograniczona lub gdy potrzebna jest duża liczba danych testowych, np. w testach wydajnościowych. Automatyzacja tego procesu czyni go opłacalnym i pozwala — w ujęciu probabilistycznym — ocenić niezawodność testowanego obiektu. Testowanie losowe pomaga również uniknąć uprzedzeń, takich jak nadmierne zaufanie do określonych fragmentów kodu. Mimo to, metoda ta ma swoje ograniczenia: może pomijać semantykę danych, przez co nie wykrywa błędów zależnych od znaczenia danych, może generować zbędne przypadki testowe, pomijać niektóre defekty, a także prowadzić do niespójnych wyników testów z powodu losowości [30]. Szczególnym przypadkiem testowania losowego jest testowanie rozmyte, które polega na *zalewaniu* oprogramowania (ang. flooding) losowymi lub zmodyfikowanymi danymi wejściowymi, co jest szeroko stosowane w obszarach związanych z cyberbezpieczeństwem, ponieważ metoda ta okazała się niezwykle skuteczna w wykrywaniu podatności w oprogramowaniu.

Generowanie następczych przypadków testowych na podstawie już istniejących oraz zdobytej wiedzy na temat testowanej funkcjonalności nosi miano testowania metamorficznego. Nowe przypadki tworzy się w oparciu o tzw. relacje metamorficzne (ang. Metamorphic Relations, MR) — są to właściwości testowanego systemu, które określają, jak zmiana danych wejściowych powinna wpłynąć na oczekiwany wynik testu. Technika ta jest stosowana w sytuacjach, gdy nie ma możliwości jednoznacznego określenia oczekiwanego wyniku testu [36–38].

Ostatnia z technik dynamicznych, testowanie na podstawie przypadków użycia lub testowanie w oparciu o scenariusz [39], jest podstawą testowania funkcjonalnego opartego na kontekście (ang. Context-Based Testing). Jest to trend, w którym zwraca się szczególną uwagę na sposób, w jaki system wbudowany będzie użytkowany. Podstawową notacją do dokumentowania przypadków użycia stanowią diagramy UML (ang. Unified Modeling Language), które opisują relację między aktorem a systemem, jednakże stosuje się często opis językiem naturalnym [40, 41]. Największą wadą takiego podejścia jest trudność w automatyzacji procesu, aczkolwiek narzędzia przetwarzania języka naturalnego (ang. Natural Language Processing, NLP), a szczególnie duże modele językowe (ang. Large Language Models, LLM) coraz lepiej radzą sobie z generowaniem przypadków testowych z tego typu dokumentacji. Możliwe jest przekazanie danych z diagramu UML lub opisu słownego do takiego narzędzia lub modelu Sztucznej Inteligencji za pomocą poleceń (ang. prompt) [42, 43].

Lista ta nie wyczerpuje stosowanych metod testowania funkcjonalnego. Dużą część stanowią techniki oparte na doświadczeniu oraz wynikające ze specyfiki danego systemu. W praktyce nie wykorzystuje się wspomnianych technik testowania pojedynczo, zamiast tego łączy się je w obrębie przypadków testowych, by zwiększyć efektywność oraz pokrycie testowe. Wpływ na rodzaj stosowanych technik ma także dobór strategii testowej, która może determinować dobór metod, które pomogą uzyskać najlepsze rezultaty.

2.5 Rola testów automatycznych i manualnych

Złożone systemy wbudowane charakteryzują się bardzo dużą liczbą przypadków testowych koniecznych do wykonania, aby zapewnić żądane pokrycie wymagań funkcjonalnych. Zwykle bardzo liczne są również testy jednostkowe oraz integracyjne, będące składową oprogramowania systemu, choć często zależy to od wewnętrznych procesów

i złożoności cyklomatycznej kodu. W ramach niniejszej pracy przeanalizowano 19 projektów, w których liczba testów funkcjonalnych mieści się w zakresie od 29 do 29 719 (mediana wynosi 1 435 przypadków testowych), co w przypadku ręcznego wykonywania każdego z nich, często po kilka razy, zajęłoby zbyt wiele czasu. Nieodzownym narzędziem stosowanym w testowaniu systemów wbudowanych staje się więc automatyzacja testów, która pozwala na wykonywanie i często także raportowanie wyników testów bez udziału człowieka w specjalnie przystosowanym do tego celu środowisku. Choć praktycznie każdy test da się zautomatyzować, to proces ten traktowany jest jako inwestycja, która wiąże się z dużymi nakładami finansowymi oraz czasowymi związanymi z tworzeniem wspomnianego środowiska oraz skryptów testów automatycznych. [27]. Koszty te amortyzują się, im częściej skrypty te są wykorzystywane [44]. W rozwijaniu oprogramowania z wykorzystaniem metodyk zwinnych, gdzie zmiany kodu są częste i wymagają testowania po każdej zmianie lub każdej nocy (ang. nightly tests), automatyzacja przynosi szereg korzyści, pozwalając oszczędzać czas, zasoby oraz nie wymaga zwykle obecności inżynierów w trakcie wykonywania testów [24].

Tworzenie środowiska do automatyzacji testów rozpoczyna się od określenia wymagań funkcjonalnych oraz niefunkcjonalnych, które powinny zostać spełnione [21]. Do programowania skryptów automatycznych stosuje się szereg języków — szczególnie wysokopoziomowych (np. C++, Python), czy graficznych (np. z wykorzystaniem narzędzi takich jak National Instruments LabVIEW lub TestStand). Całe środowisko testów automatycznych wraz z przypadkami testowymi, parametrami konfiguracyjnymi, wynikami, raportami oraz zależnościami i interfejsami pomiędzy tymi elementami określa się mianem *frameworku* [45]. W testowaniu systemów wbudowanych szczególną rolę w automatyzacji odgrywa także warstwa sprzętowa badanego systemu, która jest niezbędna do uzyskania wiarygodnych wyników. Komponenty oprogramowania uruchamiane są na docelowym sprzęcie, symulatorze, emulatorze lub systemie testowania sprzętu w pętli i kontrolowane są za pomocą wywołań funkcji w środowisku programistycznym. Odpowiedź systemu może być dodatkowo badana za pomocą urządzeń pomiarowych lub analizowana z wykorzystaniem *debuggera*. Zaletą tego podejścia jest brak wprowadzenia dodatkowego obciążenia obliczeniowego dla mikroprocesora systemu wbudowanego, gdyż testowanie obsługiwane jest po stronie środowiska programistycznego lub symulatora [46].

Automatyzacja testów umożliwia szybsze i częstsze wykonywanie testów, a co za tym idzie, daje przestrzeń, by zwiększyć także pokrycie testowe. Zwykle testy automatyczne są również bardziej niezawodne, gdyż prawidłowo napisane przypadki testowe

działające w stabilnym środowisku eliminują błąd ludzki i zapewniają większą powtarzalność. Automatyzacja testów jest także warunkiem koniecznym do implementacji środowiska ciągłej integracji (ang. continuous integration) oraz ciągłego dostarczania (ang. continuous deployment), coraz częściej stosowanych praktyk z zakresu rozwoju oprogramowania [47]. Dzięki większej elastyczności możliwe jest wykonywanie testów zarówno w dzień, jak i w nocy, co ma znaczenie również w przypadku projektów rozproszonych, prowadzonych jednocześnie w wielu lokalizacjach na świecie. Wadą tego podejścia jest ograniczona kontrola nad przebiegiem samego testu, co często prowadzi do niezrozumienia jego działania przez inżynierów analizujących raporty. Proces rozwoju automatyzacji jest kosztowny, inwestycja nie obejmuje wyłącznie stworzenia odpowiedniego środowiska i skryptów, ale także jego utrzymywanie [24].

Automatyzacja testów jest także podatna na tak zwany *„paradoks pestycydów”*, którym określa się zjawisko uodparniania się oprogramowania na powtarzane ciągle dokładnie takie same skrypty automatyczne, które przestają znajdować kolejne defekty [48]. W pewnym momencie wszystkie wykryte przez dany skrypt testowy błędy są znane lub zostały naprawione. Nie oznacza to jednak, że oprogramowanie jest wolne od defektów, a jedynie że niezmodyfikowane testy nie są w stanie ich wykryć. Wyzwaniem staje się również zmienność funkcjonalności oprogramowania wynikająca z ciągłych zmian wymagań klientów oraz wspomnianej naprawy wykrytych wcześniej błędów, która może wprowadzić nowe pomyłki w kodzie [49].

Pewne elementy systemów wbudowanych są szczególnie podatne na testowanie w sposób automatyczny. Należą do nich między innymi testy baz danych, komunikacji i cyberbezpieczeństwa, które operują na dużych i złożonych zestawach danych wejściowych i wyjściowych, bądź wymagają intensyfikacji ruchu sieciowego docierającego do systemu wbudowanego. Istotnym problemem testerskim stają się mechanizmy bezpieczeństwa, takie jak uwierzytelnianie i autoryzacja, których kluczowym elementem są generatory liczb losowych (ang. Random Number Generators, RNG) [50], a których ręczne testowanie jest praktycznie niemożliwe. Kolejną istotną grupą testów są przypadki, w których wykonanie weryfikacji lub obserwacji wymaga precyzyjnego pomiaru czasu, temperatury, pozycji, bądź innych parametrów fizycznych systemu i niedopuszczalne staje się wprowadzenie błędu bądź opóźnienia wynikającego z działania człowieka.

2.6 Stosowanie dużych modeli językowych w testowaniu

Gwałtowny rozwój dużych modeli językowych wpłynął na sposób tworzenia oprogramowania, szczególnie w obszarach wymagających zaawansowanego rozumienia i generowania języka naturalnego [51]. Systemy oparte na sieciach neuronowych typu transformer stały się przełomowymi narzędziami w tworzeniu aplikacji, które potrafią przetwarzać, rozumieć i generować odpowiedzi przypominające ludzkie. Systemy te często wykorzystują architektury generowania wspomaganego wyszukiwaniem (ang. Retrieval-Augmented Generation, RAG), które łączą potężne możliwości językowe modeli z dokładnymi, aktualnymi informacjami specyficznymi dla danej dziedziny [52].

Integracja modeli językowych z systemami automatyzacji testów niesie ze sobą bezprecedensowe wyzwania w zakresie zapewnienia jakości i testowania. W przeciwieństwie do tradycyjnych systemów o deterministycznym zachowaniu, aplikacje oparte na LLM wykazują cechy, które sprawiają, że konwencjonalne podejścia do testowania są niewystarczające. Systemy te generują niedeterministyczne odpowiedzi, zachowują się w sposób zależny od kontekstu i mogą wytwarzać tzw. halucynacje, co wpływa na ich niezawodność i wiarygodność [43]. Liczba czynników zmiennych wzrasta w przypadku systemów RAG, gdzie jakość odpowiedzi zależy nie tylko od możliwości LLM, ale również od dokładności i trafności pozyskiwanych informacji.

Generowanie wspomagane wyszukiwaniem może zostać dodatkowo wsparte za pomocą procesu dostrajania (ang. fine-tuning) w celu poprawy wydajności w zadaniach specyficznych dla danej dziedziny. Proces dostrajania obejmuje przygotowanie danych sformatowanych jako pary wejście-wyjście do uczenia nadzorowanego [53].

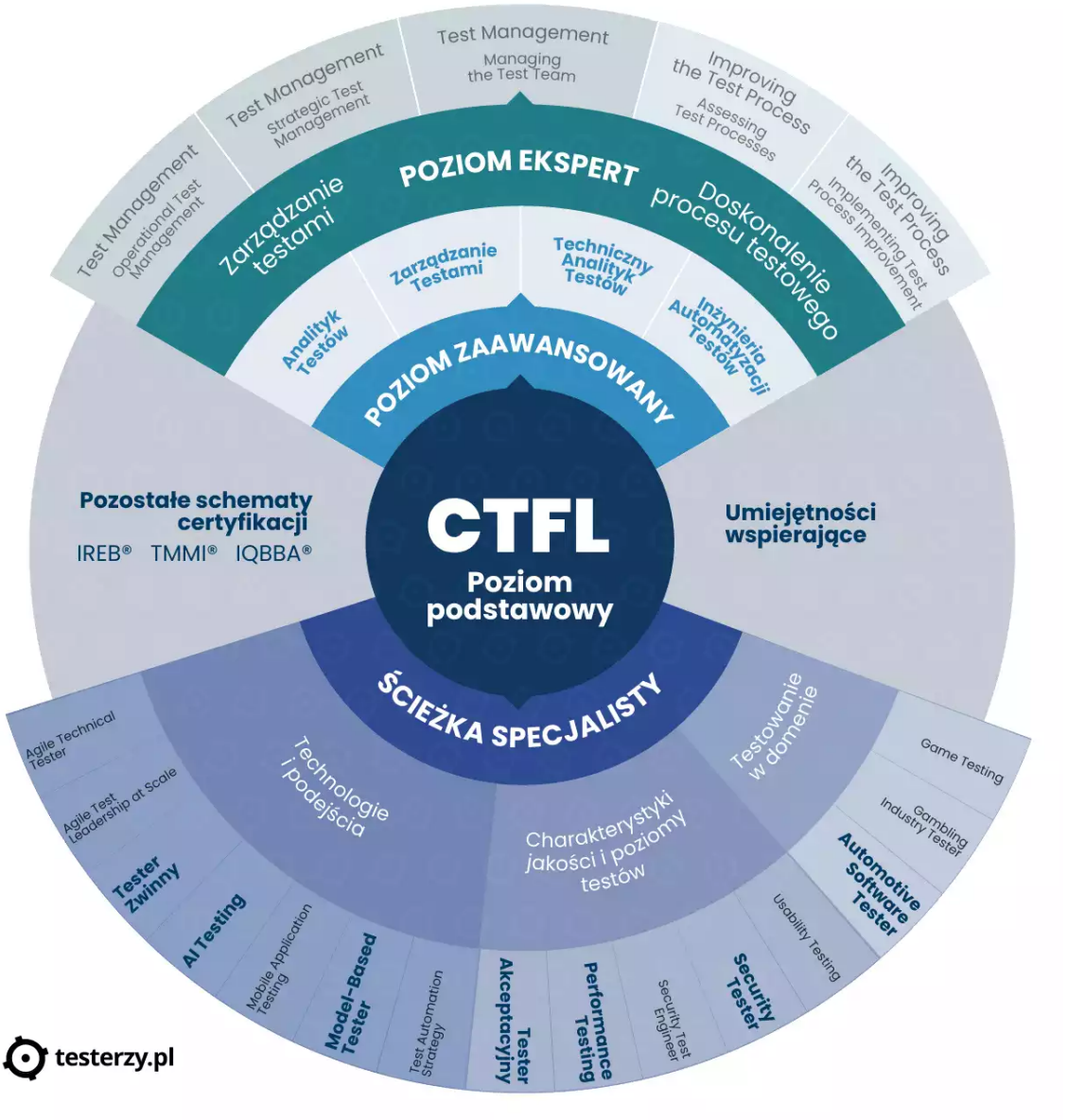
Wspomniane wady dużych modeli językowych oraz ryzyko związane z generowaniem nieprecyzyjnych lub zmyślonych odpowiedzi powodują, że użycie tych sieci neuronowych staje się ograniczone w zakresie weryfikacji i walidacji funkcjonalnej systemów wbudowanych, ze szczególnym uwzględnieniem systemów bezpiecznych. Kluczowe aspekty takie jak powtarzalność i wiarygodność testów projektowanych lub przeprowadzanych z wykorzystaniem modeli językowych nie mogą być obecnie w pełni zapewnione i zależą w dużym stopniu od dostarczonych modelowi danych uczących, sposobu uczenia, czy funkcji aktywacji [54,55]. Narzędzia te mogą być jednakże z większą pewnością stosowane do wspomagania procesu testowego, na przykład w zakresie wykrywania potencjalnych duplikatów zgłoszonych defektów [56].

2.7 Certyfikacja w testowaniu systemów wbudowanych

Międzynarodowa organizacja International Software Testing Qualifications Board (ISTQB), działająca na zasadzie wolontariatu, zajmuje się certyfikowaniem specjalistów w dziedzinie testowania oprogramowania i zapewniania jakości. Oferuje ona rozbudowany system certyfikacji, obejmujący różnorodne programy dostosowane do aktualnych standardów branżowych, który obrazuje Rysunek 2.6 [57]. Każdy z tych programów zawiera szczegółowy sylabus oraz przykładowe pytania egzaminacyjne. Materiały te mogą być również wykorzystywane jako podstawa do prowadzenia zajęć akademickich z zakresu testowania oprogramowania [58]. Do czerwca 2024 roku, ISTQB przeprowadziło 1,375 miliona egzaminów i wydało ponad 995 tysięcy certyfikacji w ponad 130 krajach [30].

Certyfikacja dla testerów oprogramowania oferowana przez tę organizację jest uznawana na całym świecie jako standard kwalifikacji testerskich na różnych poziomach zaawansowania. ISTQB oferuje specjalistyczne certyfikaty, które pomagają w zdobyciu wiedzy i umiejętności niezbędnych do efektywnego testowania oprogramowania, także dla systemów wbudowanych. Certyfikacja na poziomie podstawowym, znana jako Certyfikowany Tester Poziom Podstawowy (ang. Certified Tester Foundation Level, CTFL), zapewnia solidne podstawy w zakresie zasad testowania, procesów i technik, które są kluczowe dla każdego testera [59].

Oprócz tego ISTQB oferuje certyfikaty na poziomie zaawansowanym i eksperckim, które obejmują obszary takie jak zarządzanie testami, analityka testów, czy ulepszanie procesów testowych. Wyróżnić można też ścieżkę certyfikacyjną dla testerów zwinnych (agile) oraz ścieżkę specjalistyczną, która skupia się na testowaniu w odniesieniu do poszczególnych gałęzi przemysłu [30,59]. Certyfikaty te pomagają testerom w zdobyciu głębszej wiedzy i umiejętności, które są niezbędne do radzenia sobie także z unikalnymi wyzwaniami związanymi z testowaniem systemów wbudowanych, ale na chwilę obecną organizacja nie oferuje certyfikacji związanej ściśle z tą problematyką.



Rysunek 2.6: Certyfikacja ISTQB (Źródło: Testerzy.pl; grafika wykorzystana za zgodą autora)

2.8 Wpływ badań na certyfikację

Pomimo unikalnej architektury i implementacji, w trakcie przeprowadzania badań opisanych w dalszych rozdziałach niniejszej pracy, zaobserwowano szereg uniwersalnych wyzwań i możliwych do zastosowania technik, które są charakterystyczne dla testowania systemów wbudowanych. Należą do nich między innymi:

- planowanie właściwej strategii, która bierze pod uwagę zależności sprzętowo-programowe,
- planowanie dostępności zasobów sprzętowych,
- automatyzacja testów i budowanie *frameworków* testowych,
- znaczenie testowania opartego na doświadczeniu.

Wyniki badań nad testowaniem systemów wbudowanych oraz kolejne prace rozszerzające ich zakres mogą znacząco wpłynąć na certyfikację ISTQB, prowadząc do aktualizacji sylabusów i wprowadzenia nowych standardów branżowych. Rozwój technologii sprawia, że systemy wbudowane stają się coraz bardziej zaawansowane i pojawiają się nowe wyzwania związane z ich testowaniem. Badania mogą dostarczyć nowych metod i narzędzi, które będą musiały zostać uwzględnione w programach certyfikacyjnych ISTQB, aby testerzy byli na bieżąco z najnowszymi praktykami i technologiami. Ponadto, możliwy jest rozwój nowych specjalizacji w ramach certyfikacji ISTQB, w tym dedykowanej dla testerów systemów wbudowanych. To pozwoliłoby testerom na zdobycie bardziej wyspecjalizowanej wiedzy i umiejętności, co z kolei przyczyniłoby się do poprawy jakości i niezawodności testowanych systemów.

Rozdział 3

Charakterystyka projektu doktorskiego

3.1 Metodyka badań

Na wybór metody badawczej przyjętej do realizacji pracy doktorskiej miała wpływ empiryczna charakterystyka pracy w firmie Rockwell Automation, w której prowadzone było wdrożenie. W dziale badań i rozwoju wykorzystuje się *Scaled Agile Framework* [60], będący przedstawicielem metodyk zwinnych wytwarzania oprogramowania i sprzętu. *Framework* ten pozwala na skalowanie procesów na poziomie przedsiębiorstwa. Charakteryzuje go iteracyjne podejście do pracy, w którym przyrosty przypadają na okresy zwane interwałami planowania (ang. Planning Interval, PI). Typowy interwał obejmuje osiem do dwunastu tygodni i zawiera od czterech do sześciu dwutygodniowych iteracji (zwanych też sprintami), zakończonych spotkaniami przeglądowymi, retrospektywami oraz planowaniem kolejnych celów. W ramach PI podejmowane są decyzje dotyczące alokacji zasobów, priorytetyzacji *backlogu* oraz identyfikacji ryzyk projektowych. Aby pozostać w zgodzie z tym trybem pracy, badania zaplanowane zostały w formie eksperymentów — stopniowej modyfikacji stosowanych technik testowych i obserwacji ich wpływu na uzyskiwane wyniki testowania w odniesieniu do wyników wcześniejszych, uwzględniając starsze wersje oprogramowania oraz wcześniej rozwijane systemy wbudowane o podobnej charakterystyce. Dzięki cyklowi iteracyjnemu możliwe było inkrementalne wprowadzanie zmian i ich szybka weryfikacja.

Rozwijanie nowych produktów wykazuje cechy projektu, gdyż nosi znamiona unikal-

ności, poszczególne zadania są ze sobą powiązane, a prowadzone prace mają określony początek oraz koniec. Cykl rozwoju produktu i specyfika pracy w przedsiębiorstwie wykluczają stosowanie różnych technik testowania wielokrotnie w odniesieniu do tego samego systemu, gdyż kolejne wersje oprogramowania stale ewoluują, a presja czasu nie pozwala na powtórne testowanie tych samych funkcjonalności z wykorzystaniem innych metod. Zatem porównanie poszczególnych strategii i podejść może odbywać się wyłącznie względem innych projektów, które — choć mają wyraźne elementy podobieństwa — nigdy nie są identyczne, zatem ma ono charakter przybliżony. Należy zwrócić uwagę, że w dużych korporacjach nad testowaniem złożonego systemu wbudowanego pracują wieloosobowe zespoły rozproszone, zlokalizowane w różnych centrach inżynieryjnych na całym świecie. Kampanie testowe trwają od kilku do kilkunastu miesięcy i obejmują wykonanie nawet kilkudziesięciu tysięcy przypadków testowych, co generuje wysokie koszty związane z pracą testerów i utrzymaniem infrastruktury. W takim kontekście efektywność procesu testowego staje się istotnym czynnikiem konkurencyjności przedsiębiorstwa.

Kluczowe było przyjęcie takiej metody badawczej, która umożliwia jednoczesne uwzględnienie kontekstu projektowego, zmienności warunków realizacyjnych oraz złożoności środowiska testowego. Warunki te spełnia studium przypadku [61–64] o charakterze eksploracyjnym, umożliwiające analizę przebiegu rzeczywistych działań inżynierskich w ich naturalnym kontekście. Studium przypadku pozwala na systematyczne dokumentowanie zmian w strategii testowania oraz ocenę ich wpływu na mierzalne aspekty procesu testowego, takie jak pokrycie testowe, liczba wykrytych defektów, czas wykonania kampanii czy współczynnik automatyzacji.

Dla zapewnienia rzetelności i powtarzalności wnioskowania, dane ilościowe pochodzące z repozytoriów testów, narzędzia do zarządzania testowaniem i systemu zarządzania defektami były zestawiane z obserwacjami jakościowymi, pochodzącymi m.in. ze zbieranych metryk oraz analiz dokumentacji projektowej. Taka kombinacja pozwoliła nie tylko na identyfikację statystycznych trendów i korelacji, ale również na uchwycenie przyczynowych relacji pomiędzy modyfikacjami procesów testowych a ich skutkami w rzeczywistej pracy zespołów.

Przyjęta metoda badawcza stanowi kompromis pomiędzy rygorem naukowym a ograniczeniami praktycznymi wynikającymi z charakterystyki projektów rozwoju systemów wbudowanych. Umożliwiła ona uzyskanie wniosków o wysokim stopniu użyteczności wdrożeniowej, przy jednoczesnym zachowaniu spójności z rzeczywistością operacyjną

przedsiębiorstwa.

Przegląd literatury przedmiotu oraz analiza przedsiębiorstwa pozwala sformułować trzy główne cele badawcze:

1. Ocena strategii testowania opartej na ryzyku oraz wpływu stopnia automatyzacji testów na efektywność kampanii testowych.
2. Analiza wpływu testowania opartego na doświadczeniu, a w szczególności testów eksploracyjnych, na liczbę oraz istotność wykrytych defektów.
3. Identyfikacja czynników organizacyjnych i technicznych sprzyjających szybkiej adaptacji nowych strategii testowania w zespołach rozproszonych.

Pierwszy z celów badawczych dotyczy sprawdzenia, czy podejście do testowania oparte na analizie ryzyka oraz automatyzacja testów przenoszą się na wymierne korzyści w kontekście skrócenia czasu trwania kampanii testowej. Analiza porównawcza obejmuje projekty trwające, które podatne są na modyfikacje w tym zakresie, względem projektów zakończonych, w których testowanie oparte na ryzyku nie było stosowane, a stosunek testów automatycznych do ręcznych jest ustalony i znany. Ocenie poddane jest kryterium ilościowe, to jest czas trwania testów oraz koszt przygotowania automatyzacji względem oszczędności czasu w kolejnych rundach testowania.

Realizacja drugiego celu opiera się na zastosowaniu testowania eksploracyjnego jako elementu kampanii testowej dla trwającego projektu, aby sprawdzić, jak testowanie oparte na doświadczeniu przekłada się na liczbę znalezionych błędów w oprogramowaniu. Weryfikacja ilościowa i jakościowa obejmuje liczbę znalezionych błędów, ich typ, czas przygotowania testów eksploracyjnych oraz wykrywalność defektów, których nie wychwyciły testy formalne.

Ostatnim elementem jest identyfikacja czynników organizacyjnych i technicznych, które umożliwiają efektywną implementację nowych strategii testowania w zespołach zlokalizowanych w różnych miejscach na świecie. Realizacja celu oparta jest o pozyskane doświadczenie i ma charakter badawczo-analityczny. Jej kryterium realizacji jest zaproponowanie zestawu dobrych praktyk i wniosków.

Badaniom zostało poddanych łącznie dziewiętnaście projektów złożonych systemów wbudowanych realizowanych w Rockwell Automation w latach 2020–2024 — przemienników częstotliwości, kart rozszerzeń, sterowników programowalnych oraz modułów

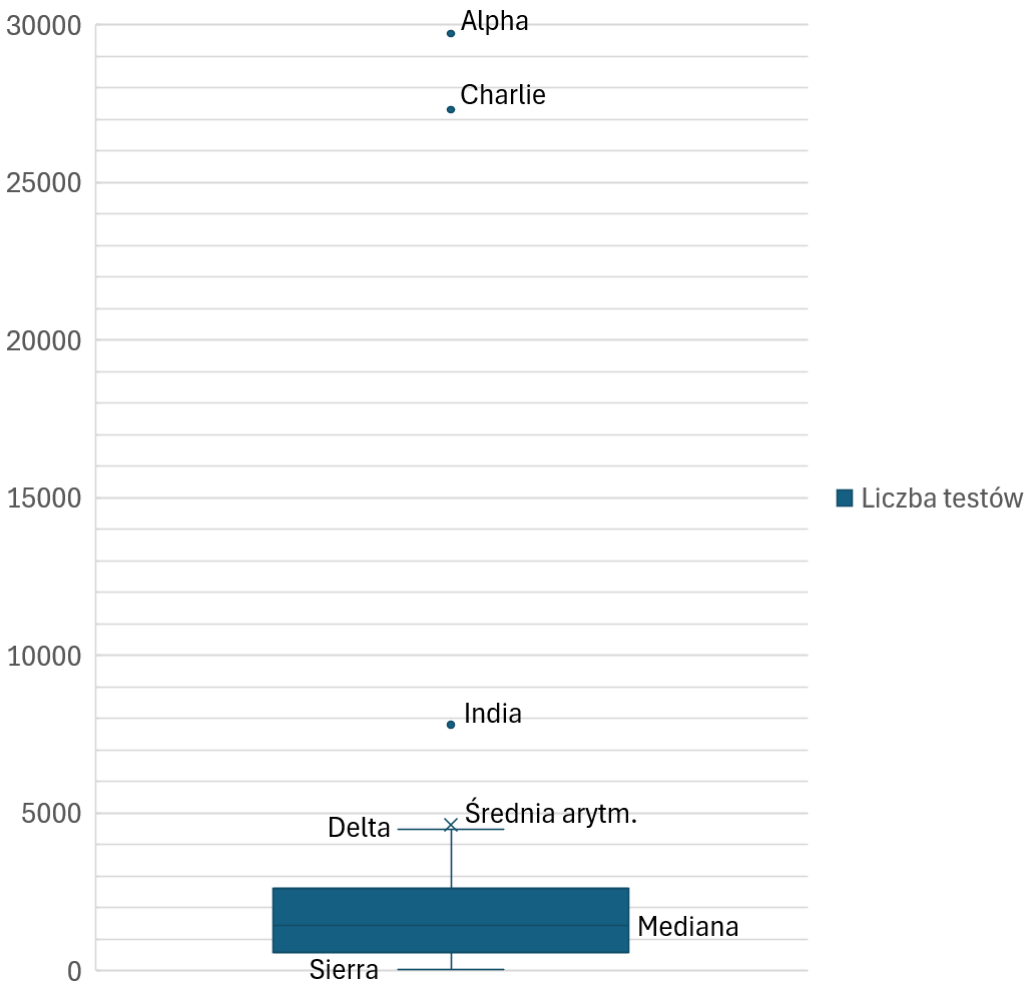
wejść-wyjść — przedstawionych w formie zanonimizowanej w Tabeli 3.1. Nadanie nazw kodowych projektom powodowane jest faktem, iż nazwy komercyjne poszczególnych produktów powiązane z danymi jakościowymi z testów funkcjonalnych stanowią tajemnicę przedsiębiorstwa i ich wykorzystanie wiązałoby się z koniecznością utajnienia wyników pracy.

Kryptonim projektu	Liczba przypadków testowych	Rozmiar
Alpha	29 719	Q5 (XL)
Bravo	2 606	Q5 (XL)
Charlie	27 317	Q5 (XL)
Delta	4 474	Q5 (XL)
Echo	1 351	Q3 (M)
Foxtrot	1 904	Q4 (L)
Golf	608	Q2 (S)
Hotel	587	Q2 (S)
India	7 812	Q5 (XL)
Juliet	1 435	Q3 (M)
Kilo	2 649	Q5 (XL)
Lima	2 246	Q4 (L)
Mike	2 166	Q4 (L)
November	92	Q1 (XS)
Oscar	1 353	Q3 (M)
Papa	223	Q1 (XS)
Quebec	72	Q1 (XS)
Romeo	1 141	Q2 (S)
Sierra	29	Q1 (XS)

Tabela 3.1: Projekty analizowane w ramach pracy badawczej

Średnia liczba przypadków testowych w projekcie to 4 620, a mediana wynosi 1 435. Aby uspójnić analizę, projekty Alpha, Charlie oraz India zostały wykluczone z dalszych rozważań ze względu na zdecydowanie większą liczbę przypadków testowych, co obrazuje Rysunek 3.1, a także niekompletne dane, co wynika z długoletniej historii tych projektów i utracenia części informacji ze względu na migracje między narzędziami.

Klasyfikacja projektów na grupy ze względu na liczbę przypadków testowych oparta została o rozkład normalny, a wartościami granicznymi przedziałów są dwudziesty, czterdziesty, sześćdziesiąty i osiemdziesiąty percentyl. Za implementację i testowanie tych produktów odpowiadały zróżnicowane zespoły, natomiast projekty o kryptonimach Quebec, Sierra, Oscar oraz Lima były podstawą prowadzonych w ramach doktoratu eksperymentów i modyfikacji strategii testowej.



Rysunek 3.1: Dystrybucja projektów ze względu na liczbę testów

3.2 Planowanie i projektowanie eksperymentów

Przed wdrożeniem działań optymalizacyjnych kampanie testowe w firmie charakteryzowały się wydłużonym czasem realizacji w stosunku do standardów branżowych. Testowanie funkcjonalne często następowało z opóźnieniem względem implementacji nowych funkcjonalności, co wynikało z następujących przyczyn:

- niestabilnego i zawodnego środowiska automatyzacji testów,
- długich i skomplikowanych przypadków testowych, które pokrywają kilka lub kilkadziesiąt wymagań funkcjonalnych,
- skryptów testów automatycznych zawierających zakodowane na stałe zmienne środowiskowe oraz parametry testowanego urządzenia,

- braku pełnego zrozumienia działania testowanego systemu przez inżynierów,
- presji czasowej nie pozwalającej na redukcję długu technicznego.

Skrypty testów funkcjonalnych tworzone były w wewnętrznie rozwijanym środowisku opartym o graficzny język programowania, zbliżony do funkcjonowania komercyjnego narzędzia National Instruments TestStand. Poszczególne kroki testu realizowane były za pomocą tzw. *aktorów*, tj. reużywalnych i parametryzowalnych elementów pełniących określone funkcje: inicjujące połączenie sieciowe z systemem wbudowanym, konfiguracyjne, pomiarowe, weryfikacyjne, obliczeniowe (w tym operacje logiczne), umożliwiające przechowywanie zmiennych i operacje na nich, tworzenie pętli, czy korzystanie z instrukcji warunkowych. Testy zorganizowane są w ramach podprogramów zwanych *workflow*, jednakże ich egzekucja możliwa była wyłącznie w całości, co w przypadku wystąpienia problemu w trakcie wykonywania testu, wymuszało powtórzenie całej, czasochłonnej procedury. Działanie środowiska, będącego aplikacją na system operacyjny Microsoft Windows, uzależnione było od dostępnych zasobów obliczeniowych komputera i systemu operacyjnego. Połączenie sieciowe z testowanym obiektem odbywało się nie bezpośrednio, a za pośrednictwem wywołania zewnętrznego oprogramowania.

Testowane systemy wbudowane, szczególnie przemienniki częstotliwości, występują w kilku odmianach sprzętowych ze względu na oferowaną moc na wyjściu, a także parametry zasilania, takie jak napięcie i częstotliwość. Ze względu na współpracę z działami inżynierskimi zlokalizowanymi w Stanach Zjednoczonych, czy Azji, parametry te powinny być odczytywane z pliku konfiguracyjnego lub danych zapisanych w pamięci testowanego systemu. Niestety praktyka pokazuje, że parametry te były przechowywane w skryptach testowych w formie stałych programowych. Co więcej, część wyników testów bezpośrednio zależy od tych parametrów, co w przypadku uruchamiania ich w innej lokalizacji skutkowało nieoczekiwanymi błędami i koniecznością wprowadzania poprawek.

Wszystkie te cechy wpływały bezpośrednio na zawodność testów oraz konieczność poświęcenia dużych nakładów pracy w celu przywrócenia ich poprawnego działania. Dodatkowo, ze względu na różnice w działaniu poszczególnych testowanych systemów, rozwijane wewnętrznie środowisko testowe wymagało tworzenia dodatkowych *aktorów*, umożliwiających interakcję, zmianę konfiguracji, czy wywołanie poszczególnych funkcji urządzeń. Kolejną wadą rozwiązania była konieczność ręcznego uruchamiania oraz raportowania wyników testów.

W trakcie gdy w firmie prowadzone były równoległe prace nad poprawą stabilności istniejących testów oraz umożliwieniem indywidualnej egzekucji poszczególnych *workflow*, w ramach części wdrożeniowej pracy doktorskiej zaproponowana została specyfikacja wymagań dotyczących nowego środowiska testowego. Określono również strategię testową dla wspomnianych wcześniej projektów Oscar oraz Lima, zorganizowaną wokół dużej liczby niewielkich i niezależnych przypadków testowych, które bezpośrednio odpowiadają poszczególnym wymaganiom funkcjonalnym testowanego systemu wbudowanego. We współpracy z zespołem stworzono prototyp środowiska testowego opartego o język Python, bibliotekę *unittest*, a także zbiór narzędzi wewnętrznych firmy, również napisanych w Pythonie[1]. Główną zaletą tego rozwiązania była szybsza oraz bardziej stabilna komunikacja z testowanym systemem, a co za tym idzie, także znacznie skrócony czas wykonywania testów. Nowe środowisko umożliwia pisanie oraz wykonywanie atomowych, tj. pokrywających pojedyncze wymaganie testów, a także automatyczne ich uruchamianie i raportowanie wyników do systemu zarządzania testami.

Część wdrożeniowa polegająca na przeprowadzaniu eksperymentów, to jest wykonywania na systemie wbudowanym testów funkcjonalnych, odbywała się przyrostowo, po każdorazowym przekazaniu przez zespół programistów wersji oprogramowania wbudowanego zawierającego stosowną funkcjonalność. Synchronizacja prac między programistami i testerami miała miejsce w ramach planowania PI oraz poszczególnych sprintów. Na podstawie wniosków cząstkowych zebranych w trakcie kolejnych eksperymentów, strategia testowa dotycząca projektów Oscar oraz Lima była w trakcie realizacji prac badawczych oraz wdrożeniowych modyfikowana poprzez włączenie elementów związanych z testowaniem opartym na ryzyku [65–67] oraz opartym na doświadczeniu [68], w szczególności testowania eksploracyjnego [69–71].

3.3 Wykorzystanie metryk w ocenie jakości testów i systemów

Jednym z kluczowych wyzwań w testowaniu funkcjonalnym systemów wbudowanych jest skuteczne mierzenie postępów oraz efektywności tych testów. Coraz więcej organizacji wymaga obecnie jednoznacznych dowodów na przeprowadzenie testów,

[1] Autorskim wkładem w tę pracę było planowanie i zarządzanie pracą w kontekście trwających projektów, formułowanie wymagań w kontekście spełnienia strategii testowej oraz wykonanie części implementacji modułów i skryptów testowych

przedstawionych w formie szczegółowych raportów [27]. Dokumenty te zawierają precyzyjne informacje dotyczące jakości testowanego produktu oraz prezentują wyniki procesu testowego. Chociaż sposób monitorowania postępów testów i raportowania może się różnić w zależności od organizacji, zazwyczaj koncentruje się on wokół pięciu głównych obszarów: ryzyk jakościowych, wykrytych defektów, przypadków testowych, pokrycia testowego oraz poziomu zaufania do produktu [72].

W kontekście testowania funkcjonalnego systemów wbudowanych, tradycyjne metody takie jak pomiar liczby linii kodu [73], analiza pokrycia mutacyjnego [74] lub stanów programu [75], a także narzędzia do mierzenia pokrycia kodu czy gałęzi, okazują się często nieskuteczne. Wynika to z faktu, że testerzy zazwyczaj nie mają dostępu do kodu źródłowego produktu. Przekonanie, że pełne pokrycie wymagań wystarcza do zapewnienia jakości, może być złudne. Może to prowadzić do sytuacji, w której produkt zostaje wprawdzie zweryfikowany, ale nieprawidłowo zwalidowany — co oznacza, że spełnia wymagania formalne, ale niekoniecznie odpowiada rzeczywistym potrzebom użytkownika. Dlatego pokrycie wymagań powinno być traktowane raczej jako dodatkowy wskaźnik, a nie jedyny wyznacznik jakości testów.

W ramach pracy doktorskiej wymagane było zaproponowanie zestawu metryk, które mają zastosowanie w testowaniu funkcjonalnym [76] oraz mogą być wdrożone w przedsiębiorstwie w celu monitorowania statusu oraz efektywności kampanii testowych. Przede wszystkim należy do nich ogólna liczba testów oraz ich status w każdym projekcie, a także podział ze względu na testy manualne i automatyczne. Pozwala to śledzić postępy w tworzeniu nowych przypadków testowych w zależności od ustalonych celów — takich jak określony poziom automatyzacji, czy liczba testów zatwierdzonych po zakończeniu przeglądu ich zawartości, a co za tym idzie, gotowych do wykonania.

Jedną z miar stosowanych do ewaluacji proponowanych zmian i strategii testowej jest współczynnik efektywności testu (ang. Test Effectiveness Ratio, TER), obliczany według Wzoru 3.1. Może on być stosowany w odniesieniu do pojedynczego przypadku testowego, jak i całych grup testów, aby porównać ich skuteczność w wykrywaniu defektów, przy założeniu, że w kodzie poszczególnych testowanych funkcjonalności występuje podobna liczba defektów.

$$TER = \frac{D_T}{D_A} \times 100\% \tag{3.1}$$

gdzie: D_T oznacza liczbę defektów wykrytych przez test, a D_A oznacza łączną liczbę

wykrytych defektów.

Inną metodą na porównanie skuteczności wdrożonych modyfikacji, na przykład względem innych projektów, jest gęstość defektów (Fault Density, FD), którą można obliczyć korzystając ze Wzoru 3.2. Pozwala ona określić, ile błędów udało się wykryć w trakcie testowania w odniesieniu do łącznej liczby wykonanych przypadków testowych. Przyjmując, że w podobnych pod względem złożoności i liczby przypadków testowych projektach oczekuje się podobnej gęstości defektów, można zastosować tę metrykę jako jedno z kryteriów wyjścia z kampanii testowej lub po zakończeniu testowania określić, czy dana strategia testowa pozwala na wykrycie większej lub mniejszej liczby błędów w odniesieniu do innych.

$$FD = \frac{D_A}{L_T} \times 100\% \tag{3.2}$$

gdzie: D_A oznacza liczbę wykrytych defektów, a L_T oznacza liczbę wszystkich wykonanych testów.

Możliwym jest również odniesienie się do efektywności pojedynczego testu w wykrywaniu danego typu defektów w kolejnych rundach testowania za pomocą indeksu efektywności testu (ang. Test Effectiveness Index, TEI, μ) określonego Wzorem 3.3. Niech T_i oznacza typ defektu. Przykładowe rodzaje defektów charakterystyczne dla testowania funkcjonalnego określa Tabela 3.2. W tym przypadku liczba defektów w *k–tym* module jest oznaczona jako $D_k^l(T_i)$, gdzie: l oznacza kolejną rundę testowania, a i oznacza rodzaj defektu. Im TEI bardziej zbliża się do wartości 1, tym bardziej efektywny jest dany przypadek testowy.

Oznaczenie	Typ defektu	Opis
T_1	Defekt funkcjonalny	Implementacja systemu wbudowanego różni się od specyfikacji wymagań funkcjonalnych.
T_2	Defekt bezpieczeństwa	Funkcjonalność bezpieczeństwa nie jest zgodna z wymaganiami bezpieczeństwa lub odkryto ryzyko związane z bezpieczeństwem.
T_3	Defekt wydajnościowy	Wydajność systemu jest niestabilna lub niezgodna ze specyfikacją.
T_4	Defekt interfejsu	Defekt związany z interfejsem użytkownika lub warstwą komunikacyjną.

Tabela 3.2: Przykładowe typy defektów w testowaniu funkcjonalnym

$$\mu_k^l = \begin{cases} \frac{1}{D_k^l |D_k^l - D_k^{l-1}|}, & \text{gdy} \quad D_k^l \neq 0 \text{ oraz } D_k^l \neq D_k^{l-1} \\ \frac{1}{D_k^l}, & \text{gdy} \quad D_k^l \neq 0 \text{ oraz } D_k^l = D_k^{l-1} \\ 1, & \text{gdy} \quad D_k^l = 0 \end{cases} \tag{3.3}$$

Wadą tych metryk jest możliwość dokładnego pomiaru dopiero po zakończeniu pewnego etapu kampanii testowej, ponieważ w trakcie jej trwania dostępne są jedynie wyniki cząstkowe. Najprostszą metodą sprawdzenia postępów w testowaniu jest liczba wykonanych testów. Pokazuje ona, jak wiele testów zostało wykonanych i ile pozostało, a prezentowana jest najczęściej w formie wykresu spalania (ang. burndown chart). Liczba ta następnie wykorzystywana jest między innymi do obliczenia gęstości defektów. Liczbę wykonanych testów można odnieść również do pokrycia specyfikacji wymagań funkcjonalnych, określając, ile wymagań zostało przetestowanych. Warto nadmienić, że pełne pokrycie wymagań nie jest kryterium wystarczającym do zakończenia kampanii testowej, gdyż pomija ono całkowicie aspekt walidacyjny testowania. Podobnie jest z określaniem czasu potrzebnego na testowanie — wyczerpanie limitu czasu nie oznacza, że proces został zakończony — choć w przypadku modyfikacji strategii testowej dostarcza nam to informacji, czy nowe podejście jest bardziej, czy mniej wydajne pod tym względem.

Możliwe jest również określenie metryk zbieranych w okresie po zakończeniu testów, gdy system wbudowany trafia do użytkowników końcowych. Należą do nich liczba niewykrytych defektów lub procent wykrytych defektów (ang. Defect Detection Percentage, DDP), który obliczany jest za pomocą Wzoru 3.4. Badanie przyczyn niewykrycia defektów w trakcie testowania jest ważnym aspektem poprawiania procesów testowych. Umożliwia ono wskazanie, na którym etapie testów defekt powinien zostać wykryty i umożliwia reakcję polegającą na dodaniu właściwych przypadków testowych lub wdrożeniu nowego typu testowania, tak by sytuacja ta nie powtarzała się w przyszłości.

$$DDP = \frac{D_A}{D_A + D_N} \times 100\% \tag{3.4}$$

gdzie: D_A oznacza liczbę wykrytych defektów, a D_N oznacza liczbę niewykrytych defektów.

Ocenie jakości podlega również automatyzacja testów. Jednym z istotnych jej parametrów jest niezawodność (ang. Test Automation Reliability, TAR), rozumiana jako

rzetelność dostarczanych wyników, określona Wzorem 3.5. Dostarcza ona informacji o udziale wyników fałszywie pozytywnych (w których test kończy się błędnie określonym negatywnym rezultatem) oraz fałszywie negatywnych (w których defekt nie został wykryty, mimo że powinien) względem całkowitej liczby skryptów testowych.

$$\mathrm{TAR} = (1 - \frac{R_F}{L_A}) \times 100\%. \tag{3.5}$$

gdzie: R_F oznacza liczbę wyników fałszywie pozytywnych i negatywnych, a L_A oznacza liczbę testów automatycznych.

Zbieranie i analiza metryk są zadaniami czasochłonnymi oraz polegającymi w dużej mierze na jakości i rzetelności dostarczanych danych wejściowych, dającymi natomiast wymierne efekty w ocenie jakości systemów i procesu testowania. Dobrą praktyką jest stosowanie narzędzi umożliwiających automatyzację zbierania metryk oraz możliwość przedstawiania czytelnych raportów, na przykład Microsoft Power BI. Równie istotne jest edukowanie zespołów oraz interesariuszy, dlaczego dane są zbierane oraz w jakim celu są wykorzystywane. Brak metryk z kolei praktycznie dyskwalifikuje z możliwości podejmowania świadomych decyzji i korekty praktyk testerskich w organizacji, gdyż nie jest możliwe dokonanie obiektywnej oceny podjętych działań.

3.4 Teza badawcza

Testowanie funkcjonalne systemów wbudowanych stanowi jedno z bardziej złożonych wyzwań we współczesnej inżynierii oprogramowania. Pomimo znacznego postępu w dziedzinie testowania oprogramowania, specyficzne charakterystyki systemów wbudowanych wprowadzają unikalne problemy, których tradycyjne metody testowania nie są w stanie w pełni rozwiązać.

Znaczącą wadą obecnych rozwiązań testowych jest zależność od sprzętu i ograniczony dostęp do niego. Testowanie oprogramowania wbudowanego wymaga często badania finalnego produktu, obejmującego zarówno sprzęt, jak i oprogramowanie. Problem pojawia się szczególnie na wczesnych etapach testowania, gdy kompletna platforma sprzętowa może nie być dostępna. Konsekwencją tego jest konieczność polegania na emulatorach i symulatorach, które nie odzwierciedlają dokładnie rzeczywistego zachowania prawdziwych urządzeń.

Ze względu na powtarzalność zdarzeń zarówno w oprogramowaniu, jak i sprzęcie,

odtworzenie defektów w testowaniu wbudowanym jest trudniejsze. Zbieranie danych reprodukcyjnych jest wyzwaniem ze względu na konieczność interakcji z dynamicznym, często niedeterministycznym środowiskiem fizycznym. Tradycyjne testy z oddzielonymi od sprzętu środowiskami symulacyjnymi nie są w stanie uchwycić dynamicznych cech tych interakcji, takich jak przerwania sprzętowe czy rzeczywiste taktowania komunikacji, co prowadzi do systematycznych odchyleń między wynikami weryfikacji a rzeczywistym zachowaniem systemu. Nieodłączny niedeterminizm systemów czasu rzeczywistego, spowodowany przez warunki wyścigowe, przerwania sprzętowe i wymianę danych z zewnętrznym kontekstem, ma poważny wpływ na powtarzalność i przewidywalność. Czynniki nieliniowe, takie jak rywalizacja o zasoby sprzętowe i opóźnienia przerwań, nie są formalnie modelowane w tradycyjnych *frameworkach* testowych, co sprawia, że defekty współbieżności, takie jak potencjalne zakleszczenia i inwersja priorytetów, są trudne do skutecznego wykrycia.

Możliwość wystąpienia defektów zarówno po stronie oprogramowania, jak i sprzętu, stanowi także istotny problem diagnostyczny. W testowaniu wbudowanym często wykrywa się wysoki wskaźnik defektów systemowych, ponieważ testowane są zarówno oprogramowanie, jak i sprzęt. Niełatwo jest określić, której warstwy systemu dotyczą, co komplikuje *debugowanie*. W takich przypadkach krytyczna jest wiedza zespołu testowego z zakresu informatyki oraz elektroniki, by pomóc wskazać źródło problemu.

Sporym wyzwaniem jest brak standaryzacji w komunikacji z testowanym systemem. Fakt, że nie istnieje jeden protokół komunikacyjny ani kanał, który można wykorzystać do komunikacji z urządzeniami wbudowanymi, wymaga znajomości i adaptacji wielu protokołów i kanałów komunikacyjnych przy testowaniu oprogramowania dla tych urządzeń. Podobnie jest z wykorzystywanymi narzędziami wspomagającymi testowanie — wiele organizacji korzysta z własnych, specjalistycznych środowisk do weryfikacji systemów wbudowanych, albo polega na generycznych narzędziach, które są dostosowywane do specyficznych potrzeb systemu. Stwarza to kolejne wyzwania i koszty związane z utrzymywaniem takich środowisk.

Także brak systematycznego podejścia do oceny efektywności metod testowych stanowi jedną z najbardziej istotnych, choć często pomijanych, wad obecnych rozwiązań w testowaniu funkcjonalnym systemów wbudowanych. Problem ten manifestuje się w kilku kluczowych obszarach, które znacząco utrudniają obiektywne porównanie różnych technik i metodologii testowych.

Brak zunifikowanych metryk umożliwiających obiektywną ocenę efektywności róż-

nych metod testowych sprawia, że porównanie wyników między różnymi organizacjami, projektami czy metodologiami staje się niemal niemożliwe. W konsekwencji ogranicza to możliwość identyfikacji najbardziej efektywnych metod, utrudnia rozwój dziedziny i hamuje wdrażanie dobrych praktyk opartych na rzetelnych danych empirycznych. W przemyśle dominują praktyki *ad hoc*, które charakteryzują się brakiem formalnego procesu i wymaganej dokumentacji. Chociaż testowanie *ad hoc* może być skuteczne w identyfikowaniu defektów, które mogą umknąć technikom formalnym, jego nieformalny charakter utrudnia systematyczne gromadzenie danych i porównywanie rezultatów. Stan praktyki w zakresie testowania różni się między organizacjami i znacznie odbiega od stanu wiedzy opisanego w literaturze naukowej. Metody powszechnie opisywane w literaturze, takie jak testowanie eksploracyjne czy analiza ryzyka, nie są używane na szeroką skalę w przemyśle.

Brak naukowego podejścia do oceny metod testowych ma poważne konsekwencje dla całej dziedziny:

- brak możliwości obiektywnej oceny postępu — bez standaryzowanych metryk trudno jest określić, czy nowe metody rzeczywiście mają przewagę nad istniejącymi rozwiązaniami,
- duplikacja wysiłków — organizacje często ponoszą koszta projektując podejście testowe nowe dla nich, ale istniejące w domenie,
- ograniczona wymiana wiedzy — brak porównywalnych wyników utrudnia dzielenie się dobrymi praktykami między organizacjami,
- trudności w optymalizacji procesów — bez obiektywnych metryk trudno jest podejmować decyzje o tym, które metody testowe są najbardziej efektywne w konkretnych kontekstach.

Bez systematycznego podejścia obszar testowania funkcjonalnego systemów wbudowanych będzie nadal cierpiał na brak obiektywnej podstawy do porównania i wyboru najefektywniejszych metod testowych, co ostatecznie wpływa negatywnie na jakość i niezawodność systemów wbudowanych. Analiza rozwiązań opisywanych w literaturze, a także wnikliwe badanie procesu testowania funkcjonalnego złożonych systemów wbudowanych w firmie Rockwell Automation, jest próbą zaadresowania tego zagadnienia i podstawą do sformułowania tezy rozprawy:

Zastosowanie hybrydowego *frameworku* testowego, opartego na synergii deterministycznej automatyzacji, systematycznej analizy ryzyka, atomizacji przypadków testowych oraz usystematyzowanego testowania eksploracyjnego, pozwala na mierzalną optymalizację procesu weryfikacji złożonych systemów wbudowanych, wyrażoną przez statystycznie istotne podniesienie wartości metryk jakościowych — w szczególności gęstości defektów FD i współczynnika efektywności testów TER — w stosunku do tradycyjnych, monolitycznych strategii testowych.

Rozdział 4

Przyjęta strategia testowania i wyniki badań

4.1 Testowanie oparte na wymaganiach

Testowanie oparte na wymaganiach to podejście, w którym przypadki testowe są tworzone na podstawie specyfikacji wymagań systemu wbudowanego. W kaskadowym modelu wytwarzania oprogramowania, specyfikacja jest tworzona w całości przed rozpoczęciem fazy implementacji oraz testowania, podczas gdy w modelu zwinnym może powstawać w sposób iteracyjny, w miarę rozwijania poszczególnych funkcjonalności systemu. Wszelkie niejasności w wymaganiach zapisanych w języku naturalnym mogą prowadzić do poważnych błędów w kolejnych fazach rozwoju projektu. Wpływ na jakość wymagań oraz przypadków testowych może mieć zaangażowanie testerów w przegląd specyfikacji. Istnieją standardy przemysłowe, definiujące szereg kryteriów jakościowych dla indywidualnych wymagań [77], które mogą służyć jako podstawa ich oceny:

- poprawność — trafność opisu potrzeb klienta,
- wykonalność — realizowalność w ramach znanych możliwości oraz ograniczeń,
- konieczność — wyrażalność wyłącznie niezbędnych oczekiwań,
- priorytet — pilność realizacji,
- jednoznaczność — jednakowość rozumienia wymagania przez użytkowników,

- weryfikowalność — możliwość utworzenia testów, demonstracji, przeprowadzenia inspekcji, lub analizy (np. symulacji, modelu),
- atomowość — niepodzielność poszczególnych wymagań i ograniczenie ich zakresu do pojedynczych cech systemu,
- niezależność od implementacji — treść pomijająca kwestie praktycznej realizacji wymagania.

Osobne kryteria jakościowe stosuje się na poziomie dokumentu specyfikacji wymagań:

- kompletność — nie pominięcie żadnego istotnego aspektu,
- spójność — nie zawieranie sprzecznych wymagań,
- modyfikowalność — możliwość kontrolowanej modyfikacji wraz z rejestrowaniem zmian,
- możliwość śledzenia powiązań (ang. traceability) — ustanowienie połączenia z zależnymi artefaktami, takimi jak źródło, cel, przypadek testowy, implementacja.

Śledzenie powiązań pomiędzy wymaganiami, a testami zapewnia możliwość odszukania artefaktów, które są merytorycznie powiązane z danym przypadkiem testowym. Dzięki temu mechanizmowi możliwe jest przeprowadzenie analizy wpływu zmiany danego wymagania na inne artefakty, określenie postępów w implementacji i przetestowaniu wymagań, a także sprawdzenie, czy wymagania nadają się do powtórnego wykorzystania (ang. reusability). Testowanie funkcjonalne jest bezpośrednio powiązane z wymaganiami funkcjonalnymi, które są uszczegółowieniem wymagań biznesowych lub wynikają z norm przemysłowych (np. ISO 26262 lub IEC 61508). Śledzenie powiązań jest najczęściej realizowane przez dedykowane oprogramowanie do zarządzania wymaganiami, a kryterium wyjścia z testowania może być osiągnięcie określonego stopnia przetestowania wymagań.

Poprawnie sformułowane wymagania stanowią także istotny czynnik techniczny sprzyjający testowaniu. Testowanie na podstawie wymagań stanowi trzon procesu testowego wykorzystywanego w firmie Rockwell Automation. Proces Rockwell Automation Product Lifecycle (RAPL) w zakresie testowania systemów (ang. Verification and Validation) wskazuje wymagania jako podstawową informację wejściową do procesu

oraz określa jako oczekiwany stopień pokrycia co najmniej wszystkich wymagań dotyczących cyberbezpieczeństwa.

Aby uniknąć nieścisłości, zmienności i problemów gramatycznych, a także umożliwić pracę przy zaangażowaniu inżynierów różnych narodowości, wymagania najczęściej spisywane są w języku angielskim [78]. W celu dalszej eliminacji niejednoznaczności, wymagania mogą być zapisywane w zapisach formalnych (np. Z, VDM) lub półformalnych (np. UML, SysML), a także w inny usystematyzowany sposób, co umożliwia tworzenie modeli zachowania systemu i automatyczne generowanie testów na podstawie tychże (ang. Model-Based Testing, MBT). Spójna notacja może obowiązywać w określonym obszarze funkcjonalnym lub kategorii wymagań. Zaletą takiego rozwiązania jest skrócenie czasu potrzebnego na tworzenie dokumentacji testów oraz skryptów testów automatycznych.

Spójna notacja i oparta o nią automatyczna generacja przypadków testowych została zastosowana jako nowy element strategii testowej wdrożonej w ramach realizacji projektu doktorskiego w zakresie testowania parametrów konfiguracyjnych badanych systemów wbudowanych o kryptonimach Oscar oraz Lima. Specyfikacja wymagań przyjęła formę ustrukturyzowanych plików .JSON opisujących parametry urządzeń w strukturze obiektowej, to jest podzielonych na klasy, instancje oraz poszczególne atrybuty. Na najwyższym poziomie hierarchii znajduje się sekcja *Common*, która zawiera ogólny wzorzec dopasowania oraz obiekt *Class* przechowujący przykładową klasę. Każda klasa może mieć wiele instancji. Instancje zawierają kolekcje atrybutów o właściwościach takich jak wartość, typ, zakresy przyjmowanych wartości, wartość domyślna, czy jednostka. Pliki, których przykładowy fragment przedstawia Listing 4.1, zostały przetworzone przez program napisany w języku Python, generujący na tej podstawie opis kroków testu oraz skrypt testu automatycznego.

```
1    {
2   "Common": {
3     "Regex": ".*",
4     "Class": {
5           "001": {
6         "#name": "ClassName",
7         "Instance": {
8           "0": {
9             "Attribute": {
10              "1": {
11                "fields": [
```

```
12                  {
13                    "value": null,
14                    "type": "UINT',␣'VARRAY:UINT",
15                    "variance": null,
16                    "defaultValue": null,
17                    "min": null,
18                    "max": null,
19                    "validValues": null,
20                    "engUnit": null
21                  }
22                ],
23                "dataClassification": null,
24                "#name": "OptionalAttributeList"
25              },
26            },
27          }
28        }
29              },
30                 }
31  },
32  }
```

Listing 4.1: Przykładowy fragment pliku .JSON opisującego obiekty, instancje i atrybuty urządzenia

Podejście to pozwoliło na zredukowanie zaangażowania testerów w odniesieniu do 437 przypadków testowych, co przy założeniu dwóch roboczogodzin na dokumentację przypadku testowego i jego automatyzację, można szacunkowo przeliczyć na 146 dni pracy inżyniera (przyjmując sześciogodzinny dzień pracy). Wygenerowane kroki testu mogą wyglądać jak na Listingu 4.2. Przypadek testowy polega na weryfikacji poprawności odczytu atrybutu *OptionalAttributeList* z klasy *ClassName* i instancji „0" przy użyciu usługi *Get Attribute Single* służącej do odczytu danych. Test obejmuje sprawdzenie, czy zwracane dane są zgodne z definicją w pliku JSON, gdzie wartość atrybutu to *null*, a typ danych określony jako *UINT* lub *VARRAY:UINT*. Oczekiwanym rezultatem jest poprawny status odpowiedzi oraz brak wartości lub pusta tablica, co potwierdza zgodność z konfiguracją. Dodatkowo weryfikowane są pola *defaultValue* i *engUnit*, które również powinny być puste, oraz reakcja systemu na próby odczytu atrybutów spoza zdefiniowanego zakresu, gdzie spodziewany jest błędny status. Test zapewnia kompleksową ocenę integralności danych, zgodności typów i obsługi błędów w komunikacji z urządzeniem.

```
1)  Odczyt wartosci atrybutu OptionalAttributeList
Kroki:
Za pomoca narzedzia Message Tool i kodu uslugi Get Attribute
    ↪ Single odczytaj dane z:
Class: 0x001 (ClassName)
Instance: 0x00
Attr: 0x01 (OptionalAttributeList)

Oczekiwany rezultat:
Status kodu: (Success)
Zwrocone dane = brak wartosci (null) lub pusta tablica, zgodnie
z definicja w JSON.

2) Weryfikacja typu danych atrybutu
Kroki:
Za pomoca Message Tool i kodu uslugi Get Attribute Single
    ↪ odczytaj typ danych dla atrybutu OptionalAttributeList.

Oczekiwany rezultat:
Status kodu: (Success)
Zwrocony typ danych = UINT lub VARRAY:UINT.

3) Wartosci poza zakresem
Kroki:
Za pomoca Message Tool i kodu uslugi Get Attribute Single sprobuj
    ↪ odczytac atrybut o numerze spoza zdefiniowanego zakresu.

Oczekiwany rezultat:
Status kodu: (Attribute not supported)
Brak zwroconych danych.

4) Weryfikacja wartosci domyslnej i jednostki inzynierskiej
Kroki:
Za pomoca Message Tool i kodu uslugi Get Attribute Single
    ↪ odczytaj pola defaultValue i engUnit dla atrybutu
    ↪ OptionalAttributeList.

Oczekiwany rezultat:
Status kodu: (Success)
Zwrocone wartosci: defaultValue = null, engUnit = null.
```

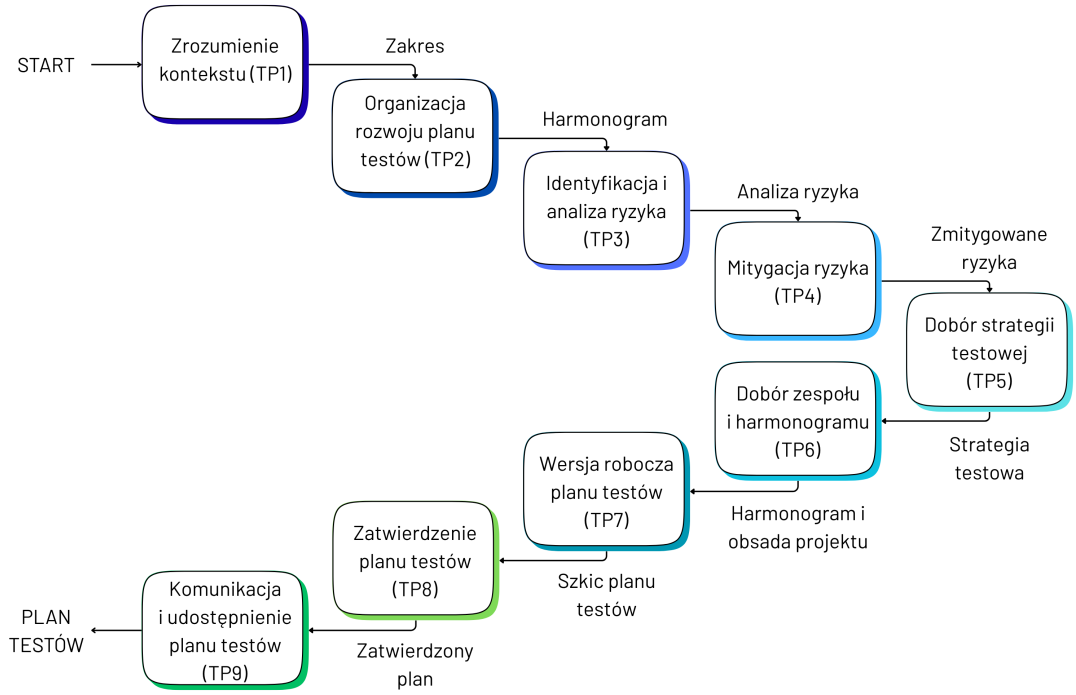
Listing 4.2: Przykładowy test wygenerowany na podstawie pliku JSON

Kolejna wprowadzona zmiana dotyczyła śledzenia powiązań między testami, a wymaganiami. W poprzednich projektach śledzenie to odbywało się na zasadzie jeden do wielu, to znaczy, że jeden test odpowiedzialny za kompleksową weryfikację danej funkcjonalności, powiązany był z wieloma wymaganiami — rekordowa liczba powiązań testu z wymaganiami przekraczała sto. Rodziło to szereg nieporozumień podczas prowadzenia analizy, które z wymagań nie zostały poprawnie zaimplementowane. Dla projektów Oscar i Lima wdrożono zmianę, w ramach której testy tworzone były możliwie atomowo, aby pokrywać jak najmniejszą liczbę wymagań. Założenie to udało się spełnić dla 99 % wszystkich testów, a w pozostałych przypadkach liczba powiązanych wymagań nie przekroczyła siedmiu.

4.2 Testowanie oparte na analizie ryzyka

Systematyczne połączenie oceny ryzyka z procesem testowania określa się mianem testowania opartego na analizie ryzyka. W tym podejściu zidentyfikowane ryzyka związane z oprogramowaniem stanowią kluczowy czynnik kierujący wszystkimi etapami procesu testowego, takimi jak planowanie, projektowanie, implementacja, wykonanie oraz ewaluacja [67]. Testowanie oparte na ryzyku ma duży potencjał usprawnienia procesu tworzenia i testowania oprogramowania, ponieważ pomaga w optymalnym wykorzystaniu zasobów oraz wspiera podejmowanie decyzji przez osoby zarządzające testowaniem [79]. Ponadto, ze względu na ograniczone zasoby i napięty terminarz projektu, testowanie wykonywane jest pod dużą presją czasu, co w konsekwencji oznacza, że można wykonać jedynie część wszystkich przypadków testowych [80]. Testowanie oparte na ryzyku dostarcza odpowiedzi, które z nich są kluczowe, a które niosą za sobą niskie ryzyko w przypadku ich pominięcia.

Norma ISO/IEC/IEEE 29119-2:2021 odnosi się wprost do testowania opartego na ryzyku jako integralnej części planowania testów, zgodnie z Rysunkiem 4.1. Kroki TP1, TP3, TP4 oraz TP5 bezpośrednio dotyczą tematów ryzyka projektowego. Kontekst określa ogólne ramy procesu oceny ryzyka i testowania. Obejmuje on źródła ryzyk — biznesowe, jakościowe, lub technologiczne. Analiza ryzyk obejmuje przypisanie im wartości będącej iloczynem prawdopodobieństwa ich wystąpienia oraz potencjalnych konsekwencji w razie gdy dane ryzyko by się zmaterializowało, co wyraża Wzór 4.1. Jeśli prawdopodobieństwo i konsekwencje wyrażone będą w skali 1–10, to ocena ryzyka będzie kształtować się w zakresie 1–100.



Rysunek 4.1: Proces planowania testów zdefiniowany w normie ISO/IEC/IEEE 29119-2:2021

$$R = P \times K \tag{4.1}$$

gdzie: R oznacza ocenę ryzyka, P oznacza prawdopodobieństwo wystąpienia ryzyka, K oznacza konsekwencje wystąpienia ryzyka.

Kolejnym etapem jest przypisanie akcji do poszczególnych ryzyk. W zależności od oceny ryzyka i kontekstu, mogą one obejmować eliminację ryzyka, dodatkowe kroki zapobiegawcze, monitorowanie, a także akceptację potencjalnych skutków wystąpienia. Decyzje o poszczególnych akcjach stanowią bazę do dobrania właściwej strategii testowej.

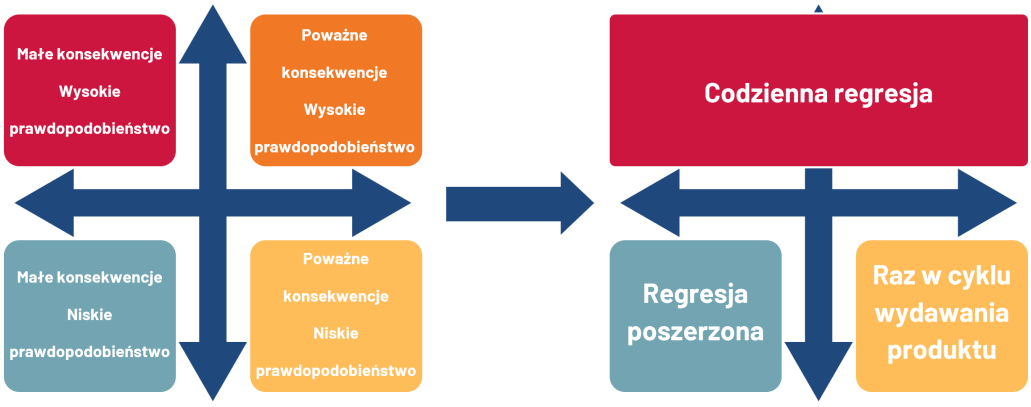
Dla omawianych projektów Oscar oraz Lima, strategia testowa była określana na etapie dokumentowania obszarowych planów testów, wchodzących w skład głównego planu testów, który obejmuje testy funkcjonalne oprogramowania wbudowanego, testy funkcjonalne oprogramowania, testy sprzętowe, testy systemowe oraz testy w zakresie cyberbezpieczeństwa. Każdy z obszarowych planów testów definiuje wysokopoziomowe scenariusze testowe, które opisują, co będzie w ich ramach testowane, nie zawierają jednak szczegółów implementacyjnych.

Bazując na ocenie ryzyka, do każdego scenariusza przypisywana jest akcja zależna od wysokości oceny, co stanowi nowy element wdrożeniowy wprowadzony w ramach projektu doktorskiego. Dla wartości poniżej 8, scenariusz nie jest pokrywany testami lub przeprowadzany jest wyłącznie przegląd kodu. Dla oceny w zakresie 8–24 funkcjonalność podlega testom manualnym, testom opartym o listę kontrolną, lub testowaniu eksploracyjnemu. Dla oceny powyżej 24 przygotowywana jest automatyzacja testu, która pozwala na regularną weryfikację funkcjonalności w trakcie regresji. Rodzaj akcji może być inny w ramach wyjątków, wynikających z kontekstu biznesowego, obowiązujących norm, czy dostępnej technologii. Tabela 4.1 przedstawia przykładową ocenę ryzyka wraz z przypisaną akcją.

Scenariusz	Ocena	Akcja	Rodzaj testu	Etap
MAJĄC [Warunek 1] ORAZ [Warunek 2] GDY [Akcja] TO [Rezultat]	P = 1–10 K = 1–10 R = 1–100	Automatyzacja	Funkcjonalny	Regresja

Tabela 4.1: Przykładowa ocena ryzyka scenariusza testowego

Podobny mechanizm przypisania akcji na podstawie analizy ryzyka zastosowany został także w przypadku testów regresyjnych. Ta sama skala użyta w odniesieniu do poszczególnych testów pozwala przypisać testy do jednej z trzech kategorii: codzienna regresja, regresja jednokrotna (zestaw testów wykonywanych raz w danym cyklu wydawania produktu), bądź regresja rozszerzona — czyli testy wykonywane, gdy pozwalają na to czas i dostępne zasoby, lub jest to wymuszone kontekstem biznesowym. Proces ten ilustruje Rysunek 4.2.



Rysunek 4.2: Analiza ryzyka w testach regresyjnych

4.3 Testowanie eksploracyjne i na podstawie doświadczenia

Tradycyjne testowanie oprogramowania opiera się na uprzednio zaprojektowanych przypadkach testowych. Alternatywą jest podejście oparte na doświadczeniu, którego przedstawicielem jest testowanie eksploracyjne (ang. Exploratory Testing, ET) [81,82]. Jego unikalną cechą jest maksymalne wykorzystanie wiedzy i inteligencji człowieka w procesie jednoczesnego uczenia się, projektowania testów i weryfikacji produktu [83,84]. Często bywa mylone z testowaniem ad-hoc, w którym tester nie ma jasno określonego celu poza interakcją z systemem. Testowanie eksploracyjne okazuje się bardzo skuteczne w wykrywaniu defektów — zwłaszcza takich, które mogłyby umknąć formalnym technikom, mają wysoką wagę, ale niską częstotliwość występowania u klienta lub są trudne do odtworzenia. Istnieje nawet hipoteza, że ET jest bardziej efektywne niż testowanie oparte na przypadkach testowych w wykrywaniu błędów funkcjonalnych, ponieważ testerzy mogą wykorzystywać swoją wiedzę do projektowania testów i identyfikowania problemów „w locie", nawet przy niewielkim doświadczeniu w testowaniu [85].

Termin „testowanie eksploracyjne" został wprowadzony przez Cema Kanera [81], a następnie rozwinięty jako dyscyplina przez Cema Kanera, Jamesa Bacha i Breta Pettichorda [82]. Testowanie eksploracyjne łączy projektowanie testów z ich wykonywaniem i koncentruje się na poznawaniu systemu poddawanego testom [86], co jest szczególnie pomocne w przypadku produktów pozbawionych pełnej specyfikacji wymagań funkcjonalnych. Istotą tego podejścia jest jednoczesne uczenie się, tworzenie projektu testów i weryfikacja produktu. Ponadto, w projektach, w których brakuje szczegółowych wymagań funkcjonalnych, testowanie eksploracyjne może być stosowane w celu zwiększenia pokrycia testami.

Choć testowanie eksploracyjne może stanowić wartościowe uzupełnienie tradycyjnego testowania, w literaturze budzi zarówno uznanie, jak i krytykę. Nie jest ono przypisane do żadnej konkretnej technologii testowania ani ograniczone do określonych cech czy aspektów — można je stosować na każdym etapie testów i w różnych implementacjach, a także łączyć z innymi technikami testowymi [87]. ET bywa często przeciwstawiane testom z zaprojektowanymi wcześniej krokami, jednak w rzeczywistości istnieje spektrum podejść — od w pełni eksploracyjnych po całkowicie skryptowe [88].

Bach i współautorzy słusznie twierdzą, że każde testowanie jest w pewnym stopniu

eksploracyjne, a eksploracja to naturalny sposób testowania [89]. Wszystkie testy wymagają znajomości testowanego systemu, jednak w przypadku ET dochodzi element trudny do uchwycenia — doświadczenie i intuicja inżyniera testów, co czasem czyni z tego podejścia swoistą sztukę. W tym kontekście testowanie eksploracyjne nie tylko opiera się na wiedzy ukrytej, ale jest procesem jej rozwijania, co w efekcie pozwala lepiej testować produkt.

W projektach Oscar i Lima testy eksploracyjne zostały wdrożone jako usystematyzowana technika [71], szczegółowo opisana w ramach ram postępowania określonych w Rozdziale 4.4. Celem zastosowania ET było zaadresowanie problemów związanych z cyklem rozwoju produktu: brakujących wymagań funkcjonalnych, zmian w zakresie projektu, małej dostępności obiektów testowych oraz zbliżającego się terminu wydania produktów na rynek. Ponadto, ze względu na nową architekturę oprogramowania oraz sprzętową, nie było możliwe przeniesienie testów z innych platform, co mogłoby być alternatywnym rozwiązaniem. Testami eksploracyjnymi objęte zostały wybrane funkcjonalności, których ocena ryzyka kwalifikowała je do tej grupy, lub wynikało to z potrzeb biznesowych:

- konfiguracja wstępna systemu (ang. startup wizard),
- automatyczne przywracania konfiguracji (ang. automatic device configuration),
- scenariusze wstrzymywania startu (ang. start inhibits),
- błędy i alarmy (ang. faults and alarms),
- testy aplikacyjne,
- współpraca z enkoderami silników.

Funkcje te zwykle wymagają bardzo rozbudowanego i czasochłonnego testowania ze względu na ich krytyczne znaczenie dla klientów oraz złożoność. Aby były w pełni operacyjne, potrzebny jest odpowiedni poziom dojrzałości sprzętu, oprogramowania i interfejsów — w przeciwnym razie dalsze testy mogą zostać zablokowane, co prowadzi do strat czasu. Już sama początkowa konfiguracja urządzenia przeznaczonego na rynek masowy wiąże się z wieloma zmiennymi i zależnościami, które należy uwzględnić. To samo dotyczy pozostałych funkcji objętych ET. To sprawia, że walidacja produktu jest wyzwaniem, ponieważ powinna wykraczać poza wymagania funkcjonalne. Testowanie eksploracyjne potrafi odpowiedzieć na większość z tych problemów — przede wszystkim

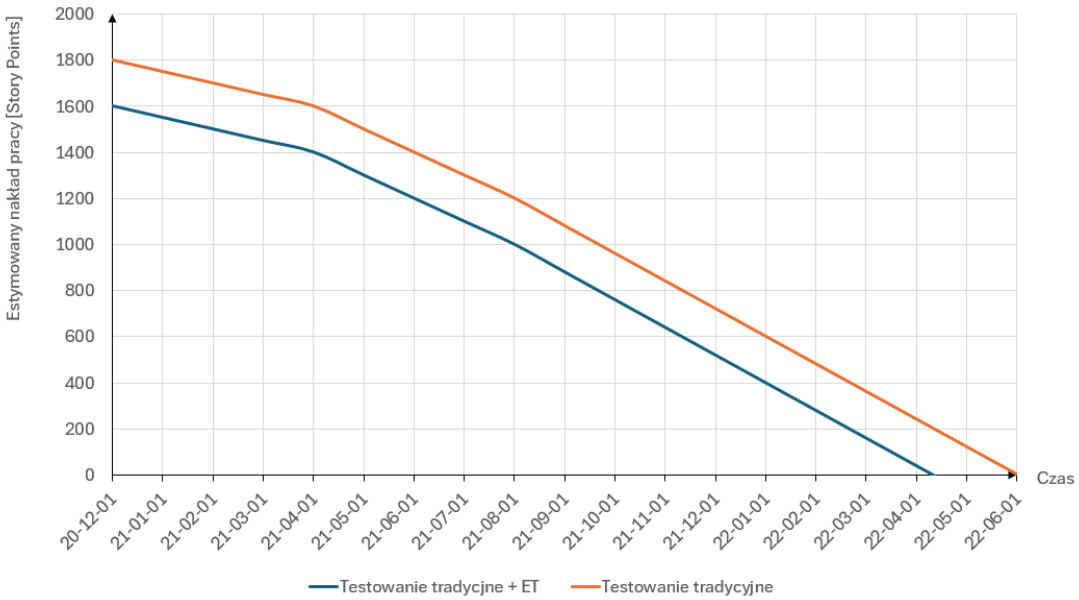
oszczędza czas potrzebny na przygotowanie skryptów testowych, a dodatkowo, dzięki koncentracji na walidacji, jest mniej zależne od stanu specyfikacji wymagań.

Różnica między tymi dwoma typami testów widoczna jest również w podejściu do weryfikacji funkcji produktu. W testach skryptowych funkcje są zwykle sprawdzane w izolacji. W analizowanym przypadku oprogramowanie ma strukturę modułową, co odzwierciedla sposób zaprojektowania wymagań funkcjonalnych, podzielonych na sekcje odpowiadające tym modułom. Oznacza to, że tradycyjne testowanie nie gwarantuje odpowiedniego pokrycia interfejsów i komunikacji między modułami. ET zakłada bardziej holistyczne podejście — w każdym scenariuszu testowane są jednocześnie liczne moduły, co pozwala skupić się również na ich wzajemnych interakcjach.

Ta filozofia stawia inżyniera testów w jeszcze bardziej zorientowanej na klienta roli niż w przypadku testów skryptowych. W trakcie ograniczonej czasowo sesji oczekuje się, że tester zapozna się z celem, warunkami i ograniczeniami danego scenariusza. Następnie może wykonywać testy w sposób zbliżony do tego, jak produkt wykorzystuje użytkownik końcowy, a podjęte kroki — opcjonalnie — rejestrować, np. narzędziem wspierającym testy eksploracyjne lub kamerą. Interakcja z testowanym urządzeniem odbywa się przez interfejs fizyczny lub, jeśli istnieje, komponent programowy umożliwiający konfigurację i obsługę produktu. W czasie sesji pożądany jest pewien zakres swobodnej eksploracji, aby dotrzeć do obszarów, które w testach skryptowych mogłyby zostać pominięte albo są trudne do zweryfikowania z powodu niejednoznacznych kryteriów.

Dla projektu Oscar wstępna estymacja nakładu pracy oraz prognoza postępu prac (z uwzględnieniem rosnącej liczby zaangażowanych inżynierów testów), zaprezentowane na Rysunku 4.3, wykazały przewidywalną oszczędność około dwóch miesięcy pracy zespołowej przy podejściu ET w porównaniu z modelem tradycyjnym. Co więcej, wygospodarowało to przestrzeń na analizę scenariuszy użycia dostarczonych przez zespół zarządzania produktem, tworząc warunki do bardziej kontekstowego testowania. Cała kampania testowa trwała ponad rok, a kryterium wyjścia stanowiło zakończenie wszystkich zaplanowanych testów. Testowanie eksploracyjne wprowadzono w ostatnich dwóch miesiącach; wykonano je na dojrzałej wersji oprogramowania wbudowanego, przez dwóch doświadczonych inżynierów testów, i zakończono w trakcie jednego, dwutygodniowego sprintu. Łącznie zrealizowano dziewięć z dziesięciu planowanych złożonych scenariuszy pokrywających wymienione wyżej funkcjonalności (ostatni został zablokowany przez czynnik zewnętrzny), co pozwoliło pokryć większość typowych przypadków

użycia.



Rysunek 4.3: Prognoza postępu prac dla projektu Oscar

Testowanie eksploracyjne dla projektu Lima zostało wzbogacone dodatkowo o elementy analizy ryzyka, gdzie dla oceny ryzyka w zakresie 8-24 funkcjonalność na podstawie decyzji menadżera testów (po konsultacji z menadżerem produktu) może podlegać testowaniu eksploracyjnemu, manualnemu lub na podstawie listy kontrolnej. Pozwoliło to na dalsze rozbudowanie scenariuszy ET oraz wykorzystanie ich w przyszłości do weryfikacji innych produktów.

4.4 Zakres testów automatycznych i manualnych

Na udział testów automatycznych i manualnych w projekcie wpływ może mieć szereg czynników, takich jak łączna liczba koniecznych do przeprowadzenia testów, konieczność ich powtarzania w przyszłości, dojrzałość produktu, możliwości technologiczne, czy czas przeznaczony na testy. W sytuacji, gdy do wykonania jest niewielka liczba testów, które w dodatku nie będą powtarzane lub będą powtarzane rzadko, można skłaniać się ku stwierdzeniu, że bardziej efektywnym będzie podejście w pełni manualne. Oznacza to, że testy będą odbywać się na podstawie określonych kroków, listy kontrolnej lub w oparciu o doświadczenie (na przykład testy eksploracyjne). W ramach analizowanych projektów, te spełniające owe kryteria zostały ujęte w Tabeli 4.2

wraz z liczbą wykonanych testów oraz liczbą znalezionych defektów.

Kryptonim	Rozmiar	Przypadki testowe	Liczba defektów
Sierra	XS	29	3
Quebec	XS	72	3
November	XS	92	2
Papa	XS	223	7
Hotel	S	587	6
Golf	S	608	6
Romeo	S	1 141	97

Tabela 4.2: Projekty kategorii XS i S

Na podstawie tych wyników, możliwe jest obliczenie współczynników efektywności testów TER, w odniesieniu do innych projektów kategorii XS i S, stanowiących dwudziesty oraz czterdziesty percentyl w rozkładzie normalnym obejmującym wszystkie analizowane projekty. Na potrzeby interpretacji wyników skorzystano ze zmodyfikowanego Wzoru 4.2, który pozwala odnieść TER do projektów w danej kategorii. Współczynniki te, przy założeniu, że ze względu na podobny poziom skomplikowania w kodzie poszczególnych testowanych produktów występuje podobna liczba defektów, mają charakter przybliżony. Wyznaczyć można również współczynnik gęstości błędów FD, dzięki czemu można dla podobnych projektów oszacować skuteczność wykonywanych testów. Wyniki obliczeń zaprezentowane są w Tabeli 4.3.

$$TER = \frac{D_P}{D_R} \times 100\% \tag{4.2}$$

gdzie: D_P oznacza liczbę defektów wykrytych w projekcie, a D_R oznacza łączną liczbę wykrytych defektów w projektach tej samej kategorii.

Kryptonim	Rozmiar	Przypadki testowe	Liczba defektów	TER [%]	FD [%]
Sierra	XS	29	3	20	10,34
Quebec	XS	72	3	20	4,17
November	XS	92	2	13,33	2,17
Papa	XS	223	7	46,67	3,14
Hotel	S	587	6	5,5	1,02
Golf	S	608	6	5,5	0,99
Romeo	S	1 141	97	88,99	8,50

Tabela 4.3: Współczynniki TER i FD dla projektów kategorii XS i S

Prowadzone w ramach doktoratu wdrożeniowego projekty Quebec oraz Sierra[1] obejmowały wyłącznie testy manualne, podczas gdy pozostałe miały zmienny udział testów automatycznych na poziomie 60-70 %. Należy jednak zwrócić uwagę, że testowanie w całości ręczne jest zasadne wyłącznie w przypadku małych projektów, których powtarzalność jest niska.

Podobną analizę można przeprowadzić dla projektów Oscar i Lima[2] w kategorii projektów złożonych, to jest obejmujących kategorie od M do XL, zgodnie z Tabelą 4.4. Projekty te charakteryzuje udział testów automatycznych na poziomie odpowiednio 50 % i 70 %. Wyższy stopień automatyzacji w projekcie Lima nie przełożył się na wyraźnie większą gęstość błędów. Wskazuje to, że zastosowanie szerszej automatyzacji ze względu na skalę i złożoność projektu, nie daje bezpośredniego przełożenia na lepszą wykrywalność błędów w oprogramowaniu.

Kryptonim	Rozmiar	Przypadki testowe	Liczba defektów	TER [%]	FD [%]
Oscar	M	1 353	204	87,18	15,08
Echo	M	1 351	11	17,19	0,81
Juliet	M	1 435	19	29,69	1,32
Foxtrot	L	1 904	18	16,07	0,95
Mike	L	2 166	32	28,57	1,48
Lima	L	2 246	62	55,36	2,76
Bravo	XL	2 606	5	6,94	0,19
Kilo	XL	2 649	37	51,39	1,40
Delta	XL	4 474	30	41,67	0,67

Tabela 4.4: Projekty kategorii M-XL

Istotną część testów w projektach Oscar i Lima stanowiły tzw. automatyczne testy promocyjne. Określały one kryteria wejścia do testowania funkcjonalnego dla kolejnych wersji oprogramowania wbudowanego w trakcie jego rozwoju poprzez weryfikację podstawowych funkcjonalności. Przejście testów oznaczało kwalifikację wersji oprogramowania na poziom dojrzałości dopuszczający testowanie funkcjonalne. Zabezpieczało to inżynierów testów przed potencjalnym zablokowaniem pracy, gdyby zaimplementowane przez programistów zmiany skutkowały awarią w kluczowych obszarach. Testy te były wykonywane w ramach środowiska ciągłej integracji CI (ang. Continuous Integra-

[1] W obu projektach rolą autora było zarządzanie testowaniem, przygotowanie planu testów (w tym dobór strategii testowej), projektowanie i wykonanie części testów oraz sporządzenie raportu końcowego. Pozostałe prace ze względu na ich zakres prowadzone były w zespole inżynierów testów.

[2] Jak wyżej.

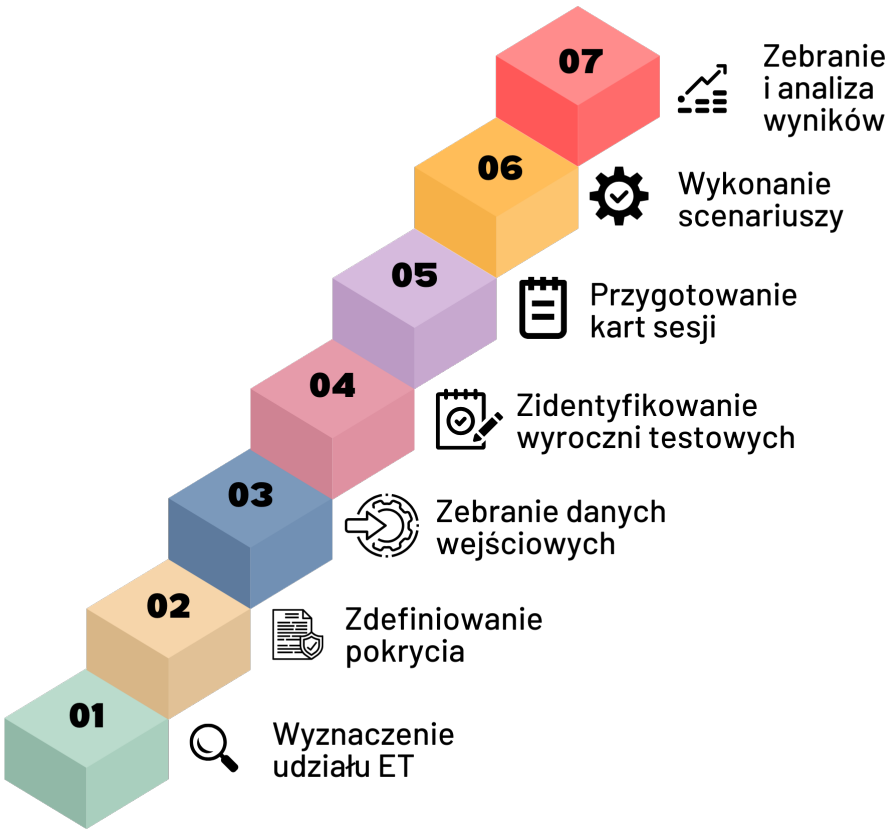
tion)[3], to jest każdorazowo podczas procesu budowania nowej wersji oprogramowania przez programistów. W skład tego zestawu testów wchodziło 361 uruchamianych regularnie przez system CI skryptów, których liczba powtórzeń w trakcie całej kampanii testowej sięgała nawet kilkuset razy. Osiągnięcie podobnych rezultatów w przypadku testowania ręcznego nie byłoby możliwe. Wykluczając jednak sporadyczne przypadki, w których dostarczona przez programistów funkcjonalność zawierała błędy wpływające znacząco na pracę całego urządzenia, testy te nie były odpowiedzialne za znalezienie dużej liczby usterek, co wynika ze zjawiska paradoksu pestycydów.

Z punktu widzenia testów manualnych, szczególną grupę stanowią testy eksploracyjne. Wyznaczone ramy postępowania ET [71], opracowane w ramach realizacji projektu doktorskiego i zastosowane w projektach Oscar i Lima, zostały zilustrowane na Rysunku 4.4 i obejmowały następujące kroki:

1. Wyznaczenie udziału testów eksploracyjnych.
2. Zdefiniowanie pokrycia.
3. Zebranie danych wejściowych.
4. Zidentyfikowanie wyroczni testowych.
5. Przygotowanie kart sesji testów eksploracyjnych:
   (a) wyznaczenie celów testów,
   (b) przygotowanie opisu,
   (c) określenie konfiguracji środowiska testowego,
   (d) wyznaczenie ram czasowych,
   (e) określenie stosunku trzymania ram testu do swobodnej eksploracji.
6. Wykonanie scenariuszy.
7. Zebranie i analiza wyników.

Pierwszym krokiem jest określenie proporcji testów eksploracyjnych w ramach danego projektu. W przypadku projektów powtarzalnych lub o wysokim stopniu podobieństwa, rekomendowane jest stosowanie podejścia skryptowego, które umożliwia szeroką

[3]Ciągła integracja to proces, który polega na częstym i regularnym włączaniu dostarczonych funkcjonalności w kodzie oprogramowania do repozytorium i automatycznej weryfikacji zmian poprzez zbudowanie projektu i wykonanie określonego zakresu testów.



Rysunek 4.4: Ramy postępowania testowania eksploracyjnego

automatyzację testów regresyjnych. Natomiast w przedsięwzięciach unikalnych, charakteryzujących się wysoką zmiennością lub brakiem pełnej specyfikacji wymagań, testowanie eksploracyjne może stanowić dominującą strategię. Ze względu na wspomniany paradoks pestycydów, uzasadnione jest włączenie elementów ET w niemal każdym kontekście. Po zdefiniowaniu skali ET, zakres pokrycia powinien zostać określony na podstawie kryteriów takich jak: stopień znajomości urządzenia lub funkcji, możliwość automatyzacji, dostępność dokumentacji oraz zasoby kadrowe.

Pomimo niższego stopnia formalizacji, ET wymaga odpowiedniego przygotowania. Konieczne jest zgromadzenie danych wejściowych w postaci dokumentacji, wymagań, scenariuszy użycia oraz wiedzy eksperckiej. Etap ten jest kluczowy dla opracowania kart sesji testów eksploracyjnych (ang. test charters), które precyzyjnie definiują ce-

le oraz identyfikują potencjalne ryzyka. W celu jednoznacznej oceny wyników należy określić wyrocznie testowe (ang. test oracles), to znaczy jasne kryteria lub źródła informacji dotyczące oczekiwanych, prawidłowych wyników testów. W przypadku ET jest to bardziej złożone niż w testach skryptowych, gdzie kryteria te są zazwyczaj jasno zdefiniowane i wynikają wprost z wymagań funkcjonalnych.

Po zakończeniu przygotowań opracowuje się kartę sesji, która dostarcza testerowi niezbędnych informacji przy zachowaniu minimalizmu w porównaniu z pełnym scenariuszem testowym. Kluczowym elementem jest zestaw celów uznanych za krytyczne dla danego testu. Opis uzupełniający może zawierać kontekst, intencje oraz informacje pomocnicze. Zaleca się również określenie konfiguracji środowiska testu (ang. test target) oraz ustalenie ram czasowych wraz z proporcją pomiędzy realizacją celów a tzw. „swobodną eksploracją" (ang. free roam). Takie podejście zapewnia dyscyplinę, a jednocześnie umożliwia eksplorację w celu identyfikacji anomalii. Brak zdefiniowanych kroków testowych stanowi fundamentalną różnicę względem testów skryptowych, pozostawiając testerowi swobodę w doborze ścieżki. Przykładową kartę sesji przedstawia Tabela 4.5.

Karta sesji ET — Konfiguracja początkowa	
Ramy czasowe	2 godziny
Opis	Przetestować kreator konfiguracji początkowej. Środowisko testowe: zestaw biurkowy lub stanowisko testowe. Stosunek ram testu do swobodnej eksploracji: 70:30.
Karta sesji	1. Przygotowanie projektu. 2. Nawiązanie połączenia z urządzeniem. 3. Testowanie kreatora konfiguracji początkowej.
Cel	1. Upewnienie się, że klient może korzystać z kreatora do stworzenia konfiguracji początkowej. 2. Zarejestrowanie kroków w celu utworzenia skryptu testowego.

Tabela 4.5: Przykładowa karta sesji ET

Realizacja sesji ET może być wspierana narzędziami rejestrującymi przebieg testów, co umożliwia późniejsze odtworzenie kroków, przekształcenie wyników w testy skryptowe oraz ułatwia reprodukcję defektów. Ostatnim etapem jest analiza i raportowanie wyników w oparciu o wcześniej zdefiniowane wyrocznie testowe lub inne kryteria weryfikacji i walidacji.

Efektywność ET jest w dużej mierze determinowana przez jakość przygotowania. W sytuacji braku jednoznacznych wymagań funkcjonalnych konieczne jest przeprowadzenie analizy w celu identyfikacji kluczowych obszarów systemu, określenia wyników testów oraz źródeł informacji. Niewłaściwie ukierunkowane ET może generować nadmierne koszty w stosunku do korzyści, co wynika również z elementu losowości w tym podejściu. Ryzyko to można ograniczyć poprzez stosowanie ram czasowych oraz świadomego bilansowania proporcji pomiędzy realizacją celów a eksploracją obszarów budzących zainteresowanie inżyniera testów.

4.5 Ewaluacja efektywności testów

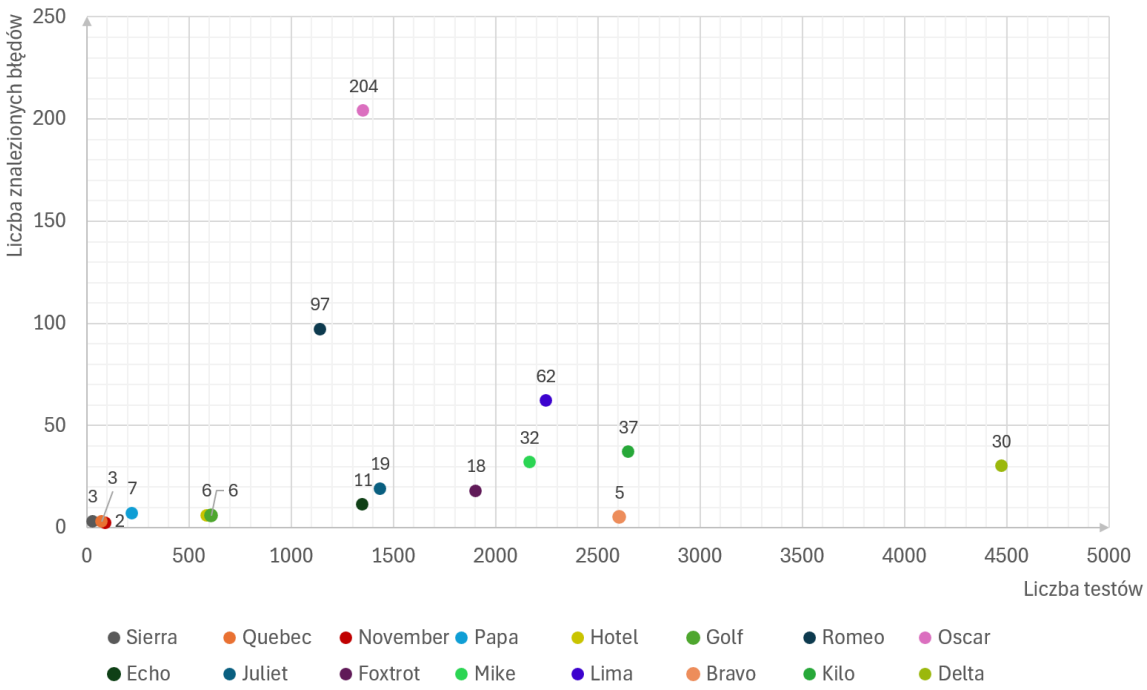
Efektywność testowania stanowi kluczowy element zapewnienia jakości w złożonych systemach wbudowanych. Można ją ocenić na wielu płaszczyznach — od liczby znalezionych defektów, poprzez gęstość błędów (FD), aż po wskaźnik efektywności testów (TER) rozumiany jako odsetek wykrytych błędów w stosunku do porównywalnych projektów. Analizę porównawczą efektywności dokonano na podstawie projektów Quebec, Sierra, Oscar oraz Lima, badając je pod kątem ilościowym i jakościowym. Punktem odniesienia są pozostałe projekty, dla których zebrano dane, przedstawione w zbiorczej Tabeli 4.6.

Analizowane projekty obejmują pełne spektrum rozmiarów, od bardzo małych, takich jak Sierra, Quebec, November i Papa, do bardzo dużych, takich jak Bravo, Kilo i Delta. Uzyskane wartości wskazują na znaczną rozbieżność w praktykach testowych, od projektów niemal pozbawionych realnej skuteczności (np. Bravo), do projektów z dużą liczbą defektów (Lima, Romeo). Zjawisko to obrazuje zależność liczby znalezionych błędów od liczby wykonanych testów widoczna na Rysunku 4.5.

Rysunek 4.6 ilustruje ujemny trend — wraz ze wzrostem liczby testów maleje gęstość defektów. Korelacja liniowa jest umiarkowanie ujemna ($r = -0,3204$), i nieistotna statystycznie ($p \approx 0,2263$), natomiast korelacja rangowa ($\rho = -0,5294$) istotna statystycznie ($p \approx 0,0350$), co wskazuje na monotoniczny spadek gęstości defektów wraz ze

Kryptonim	Rozmiar	Przypadki testowe	Liczba defektów	TER [%]	FD [%]
Sierra	XS	29	3	20	10,34
Quebec	XS	72	3	20	4,17
November	XS	92	2	13,33	2,17
Papa	XS	223	7	46,67	3,14
Hotel	S	587	6	5,5	1,02
Golf	S	608	6	5,5	0,99
Romeo	S	1 141	97	88,99	8,50
Oscar	M	1 353	204	87,18	15,08
Echo	M	1 351	11	17,19	0,81
Juliet	M	1 435	19	29,69	1,32
Foxtrot	L	1 904	18	16,07	0,95
Mike	L	2 166	32	28,57	1,48
Lima	L	2 246	62	55,36	2,76
Bravo	XL	2 606	5	6,94	0,19
Kilo	XL	2 649	37	51,39	1,40
Delta	XL	4 474	30	41,67	0,67

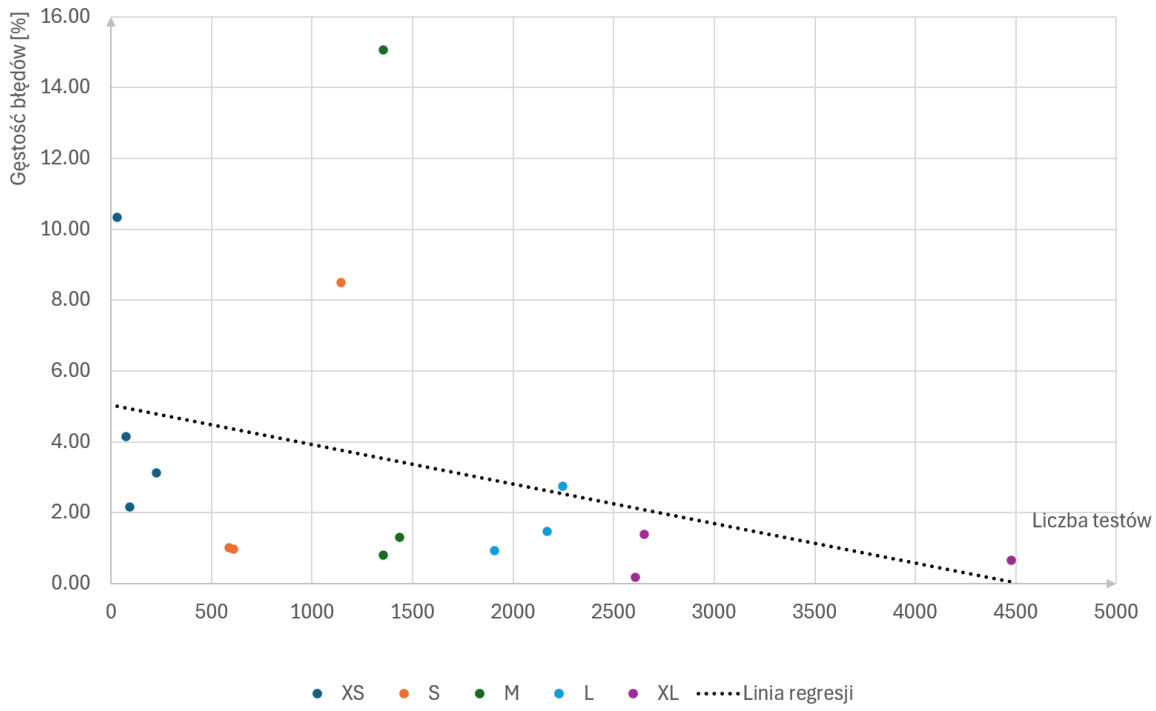
Tabela 4.6: Tabela zbiorcza projektów



Rysunek 4.5: Zależność liczby znalezionych defektów od liczby wykonanych testów

wzrostem liczby testów, choć zależność nie jest ściśle liniowa (stąd korelacja Pearsona nie jest istotna). Większy nakład testowy, rozumiany przez większą liczbą przepro-

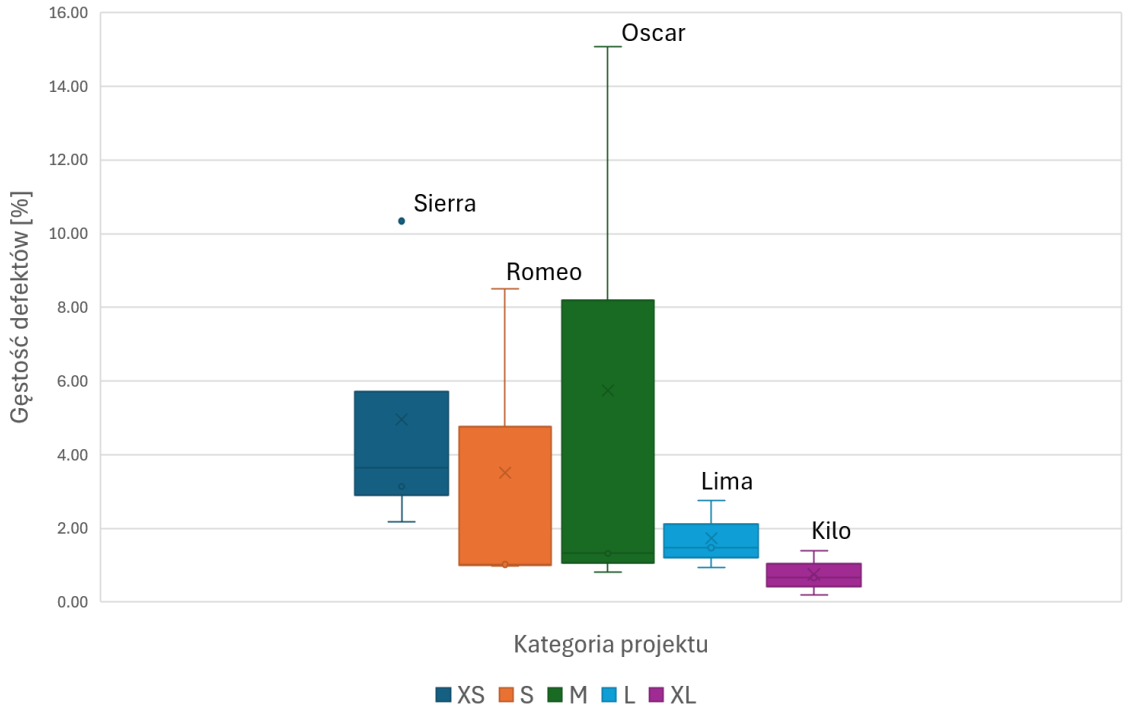
wadzonych testów, wiąże się z niższą gęstością defektów. W myśl efektu malejących przyrostów, wzrost liczby testów po przekroczeniu pewnego progu poprawia jakość bardziej stopniowo, niż liniowo, przy założeniu, że nad projektem pracują programiści o tym samym lub zbliżonym poziomie pisania jakościowego kodu.



Rysunek 4.6: Zależność gęstości błędów od liczby wykonanych testów

Projekty o wyższej liczbie testów częściej charakteryzują się niższą zmiennością w zakresie efektywności testowania, choć związek nie ma charakteru prostoliniowego i występują wyjątki (m.in. projekty o dużej liczbie testów i paradoksalnie niskim TER — Bravo). Szczególnie projekty takie jak Romeo stanowią kontrast pokazujący, jak duże mogą być różnice w przypadku projektów małych, w których występuje relatywnie wyższy udział przypadków testowych wykrywających defekty. Zjawisko to może ujawniać nierównomierne pokrycie testowe albo większą zmienność jakościową, co uwidocznione jest na Rysunku 4.7. Ponadto wraz ze wzrostem rozmiaru projektu, jednocześnie spada średnia gęstość defektów.

Testowane w pełni manualnie Sierra i Quebec znajdują dużo defektów względem liczby wykonanych testów, jednak Papa ze względu na większą liczbę testów osiąga lepszy współczynnik TER. Można więc przyjąć, że testowanie w pełni manualne, choć bardziej efektywne od automatycznego, nie nadąża za efektem skali i ograniczone jest

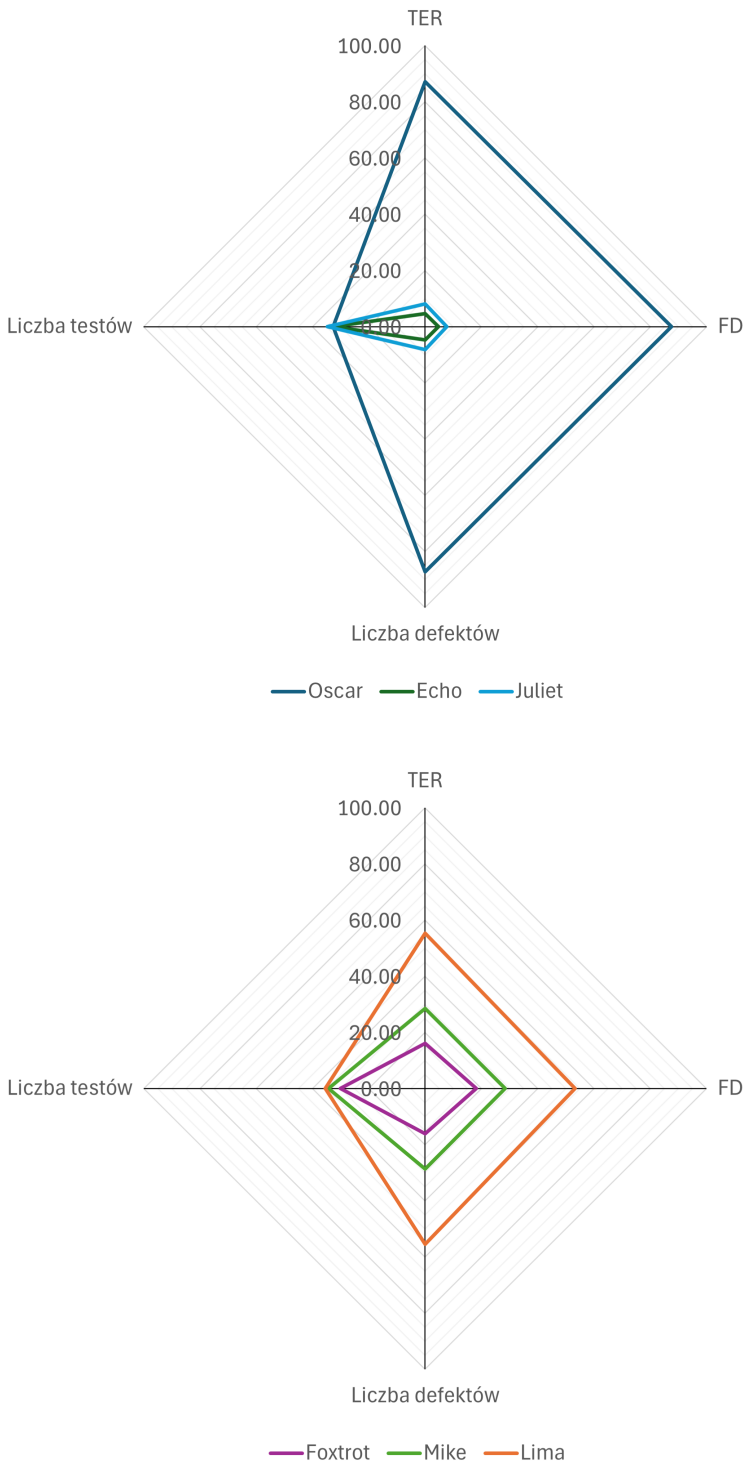


Rysunek 4.7: Dystrybucja gęstości błędów w poszczególnych kategoriach projektów

możliwą do wykonania w określonym czasie liczbą testów. Oscar i Lima dominują w swoich grupach wielkości zarówno pod względem efektywności testów, jak i gęstości błędów, stanowiąc projekty wzorcowe. Wdrożone praktyki testowe oraz strategia wydają się dobrze dobrane i sugerują wysoką dojrzałość, należy więc utrwalić je jako wytyczne procesowe. Rysunek 4.8 ilustruje, że w swoich kategoriach rozmiaru projektu Oscar i Lima wykazują wyraźnie lepsze wyniki w zakresie liczby znalezionych defektów, gęstości błędów (FD) i efektywności testów (TER).

Liczebność próby ($N = 16$) w pewnym stopniu ogranicza analizę i zastosowanie metod statystycznych. Pojedyncze przypadki odstających projektów (Bravo, Romeo) mogą nadmiernie wpływać na wnioski. Zagregowane metryki FD i TER utrudniają także rozróżnienie wpływu krytyczności znajdowanych defektów, gdyż ich ogólna liczba nie zawsze przekłada się na istotność, bądź częstotliwość występowania związanych z nimi usterek.

W ewaluacji efektywności testów nie sposób pominąć zagadnienia testów eksploracyjnych. W projekcie Oscar wykonano 9 scenariuszy testów eksploracyjnych, które znalazły 6 defektów, w porównaniu do 198 błędów znalezionych przez pozostałe 1 344 testów skryptowych, w tym 5 błędów znalezionych w trakcie wykonywania 361 te-



Rysunek 4.8: Znormalizowane porównanie projektów Oscar i Lima względem pozostałych projektów w kategoriach M i L

stów automatycznych w środowisku ciągłej integracji. Analiza defektów znalezionych w trakcie ET udowodniła, że wykrycie 5 z nich ze względu na niską odtwarzalność i częstotliwość występowania, byłoby niemożliwe przez testy automatyczne lub podążające za ściśle określonym scenariuszem. Istotność tych defektów wyznaczono na poziomie wymagającym naprawy w ramach tego samego cyklu wydawniczego produktu (ang. release cycle).

Na podstawie zrealizowanych kart sesji testów eksploracyjnych opracowano dziesięć procedur testowych, przy czym jedna z kart została rozdzielona na dwa przypadki testowe. Dwa z tych przypadków zostały włączone do zakresu testów w środowisku ciągłej integracji w kolejnych wydaniach produktu i poddane automatyzacji. Jeden przypadek został dodany do finalnego zestawu testów regresyjnych wykonywanych na oprogramowaniu w wersji kandydującej do wydania. Zmiany te zostały uwzględnione w projekcie Lima.

Szacunki nakładu pracy zostały potwierdzone w rzeczywistym przebiegu procesu testów eksploracyjnych, obejmującym przygotowanie, wykonanie oraz zarządzanie wynikami. Lekka struktura oferowana przez ET okazała się skuteczna w kontekście sztywnego harmonogramu, podczas gdy część procesu oparta na testach skryptowych napotkała opóźnienia wynikające z niedostatecznej jakości wymagań funkcjonalnych, co spowodowało przesunięcie względem estymacji przedstawionych na Rysunku 4.3 o ponad trzy tygodnie. Podsumowanie wyników ET w projekcie Oscar uwzględnione zostały w Tabeli 4.7

Rodzaj testów	Liczba testów	Defekty	Wym. naprawy	Czas trwania
Testy eksploracyjne	9	6	5	2 tygodnie
Testy skryptowe	1 344	198	79	68 tygodni
Razem	1 353	204	84	70

Tabela 4.7: Podsumowanie testów eksploracyjnych w projekcie Oscar

Wyniki należy interpretować biorąc uwagę niewielką skalę eksploracji w porównaniu do tradycyjnego testowania w oparciu o scenariusz oraz wykonywanie testów eksploracyjnych po zakończeniu testów skryptowych, w końcowej fazie kampanii testowej. Mimo tego, technika ta wykazała zdolność do ujawniania defektów o wysokiej istotności dla klienta, które z dużym prawdopodobieństwem umknęłyby w standardowym procesie testowym. Zarejestrowany przebieg działań inżynierów testów umożliwił opracowanie procedur testowych rozszerzających portfolio dostępnych przypadków —

część z nich została włączona do środowiska ciągłej integracji, a inne do zestawu testów regresyjnych. Co istotne, karty sesji mogą być formułowane w sposób niezależny od konkretnego urządzenia, dzięki czemu zachowują przenośność pomiędzy produktami o zbliżonych własnościach, lecz odmiennych implementacjach. Taka przenośność i możliwość ponownego użycia stanowią istotną przewagę z perspektywy ekonomii czasu i wysiłku zespołu.

W niniejszym kontekście szczególnie wartościowe jest rozpatrywanie zarówno liczby wykonanych testów, jak i czasochłonności poszczególnych typów testowania [76]. Mając na uwadze te aspekty, testowanie eksploracyjne zapewnia porównywalne — choć mniej dogłębne — pokrycie w znacznie krótszym czasie oraz przy mniejszej liczbie przypadków testowych, które należy zaprojektować i utrzymywać. Dzięki ramom czasowym oraz jednoznacznie zdefiniowanym celom, estymacje ET mogą być formułowane z wysoką dokładnością, a zarządzanie całością procesu pozostaje nieskomplikowane.

W przypadku złożonych, wielkoskalowych systemów wbudowanych, realizacja ET oparta na scenariuszach użycia bywa czasowo blokowana przez klastry drobnych defektów. Z tego względu rekomenduje się wykonywanie większości scenariuszy na bardziej dojrzałych etapach rozwoju produktu, kiedy wymagane interakcje licznych elementów systemu są stabilniejsze. W odróżnieniu od tego, testy skryptowe mają z natury charakter bardziej izolowany, co ogranicza ryzyko blokad wynikających z wielomodułowych zależności, ale jednocześnie redukuje zdolność do ujawniania problemów na styku komponentów.

Rozdział 5

Analiza i optymalizacja procesu testowania

5.1 Automatyzacja testów

Przed rozpoczęciem projektów Oscar i Lima, automatyczne testy funkcjonalne powstawały w wewnętrznie rozwijanym środowisku Test Automation Framework (TAF). Programowanie odbywało się w języku graficznym poprzez tworzenie sekwencji kroków — definiowanie przebiegu takiego testu było czytelne i możliwe do utrzymania w zespołach o zróżnicowanych kompetencjach, gdyż tak naprawdę nie wymagało umiejętności programowania w żadnym języku. Poszczególne kroki mogły być realizowane przez reużywalne, parametryzowalne komponenty (tzw. *aktorów*) pełniące określone role, na przykład inicjację połączenia sieciowego z systemem wbudowanym, konfigurację testowanego obiektu, wykonywanie pomiarów, weryfikację wyników, czy obliczenia. Dostępne były również konstrukcje sterujące przebiegiem, takie jak pętle oraz instrukcje warunkowe, które umożliwiały budowę złożonych scenariuszy testowych i elastyczne rozgałęzianie ścieżek wykonania. Środowisko umożliwiało także wsparcie dla wielowątkowości, chociaż funkcja ta nie była powszechnie używana.

Ze względu na brak możliwości selektywnego wykonywania pojedynczych fragmentów w razie wystąpienia problemu podczas przebiegu testu, konieczne było ponowne uruchomienie całej, często czasochłonnej procedury. Wydajność i stabilność środowiska były mocno zależne od dostępnych zasobów sprzętowych i systemowych (m.in. moc obliczeniowa, pamięć operacyjna) oraz kondycji systemu operacyjnego. Nawiązanie po-

łączenia sieciowego z testowanym obiektem wymagało wywołania zewnętrznego oprogramowania i wprowadzało zależność od komponentów spoza samego środowiska testowego. Część testów wymagała także wywołania programu sterownika PLC napisanego w języku drabinkowym LD (ang. Ladder Diagram), również za pomocą zewnętrznego programu. Każdorazowe ładowanie projektu z programem sterownika i niestabilność środowiska powodowały, że testowany obiekt po zakończeniu testów lub ich przerwaniu mógł znajdować się w stanie nieustalonym i wymagać ręcznej interwencji inżyniera testów w celu przywrócenia do stanu początkowej konfiguracji.

Pierwszą próbą zaadresowania tych problemów było wprowadzenie kryteriów stabilności oraz niezawodności testów. Celem było całkowite wyeliminowanie testów raportujących wyniki fałszywie pozytywne lub negatywne oraz losowo niestabilnych. W tym celu wprowadzono kryteria rzetelności, określone w Tabeli 5.1. Proces analizy testów obejmował opcjonalne *debugowanie* testu (jeśli okazało się konieczne) oraz kontrolę niezawodności poprzez wielokrotne uruchomienie skryptu w pętli. Jeśli dany test weryfikował funkcjonalność, na którą wpływał znany i zgłoszony defekt oprogramowania, to punkty weryfikacyjne powinny zgłaszać ostrzeżenie zamiast usterki, wraz z identyfikatorem anomalii w systemie raportowania tak, aby umożliwić późniejszą aktualizację. Liczbę powtórzeń testu ustalał inżynier testów biorąc pod uwagę czynniki takie jak czas wykonania testu, czy konieczność nadzorowania urządzenia w trakcie wykonywania procedury.

Minimalna liczba stabilnych uruchomień
10/10[1]
25/25
48/50
95/100

Tabela 5.1: Kryteria stabilności testów

Z punktu widzenia środowiska ciągłej integracji, wprowadzenie standardów niezawodności dla istniejących skryptów oraz stosowanie ich przy tworzeniu nowych, ma kluczowe znaczenie dla stabilności i powtarzalności całego procesu. Reguły te wymuszają na testerach tworzenie automatyzacji, która zapewnia możliwość uruchomienia niezależnie od pozostałej części kodu, nie wpływa na kolejne fragmenty skryptów testowych ani ich późniejsze wykonania, jak również nie jest podatna na wpływ testów poprzedzających i może działać niezależnie od konfiguracji urządzenia. Takie podej-

[1]W przypadku prostych testów.

ście zwiększa elastyczność, ułatwia utrzymanie kodu oraz minimalizuje ryzyko błędów propagujących się pomiędzy testami, co jest fundamentem skutecznej automatyzacji procesów.

Podobne problemy dotyczyły testów pisanych w Ladder Diagram i wykonywanych przez sterownik PLC. Stosowany *framework* zapewniał niezbędne interfejsy komunikacyjne, przesyłając polecenia sterujące oraz odbierając ich rezultaty, i działając jako warstwa infrastrukturalna, regulując sposób organizacji przypadków testowych, zapisywanie wyników oraz raportowanie. Istotnym założeniem dla tego typu testów, utrzymywanym w kolejnych generacjach rozwoju, była kompatybilność wsteczna. Dzięki temu rożne wersje programów sterownika mogły współistnieć, a ewolucja narzędzia nie blokowała ciągłości prac oraz ponownego wykorzystania istniejących skryptów. Niestety na potrzeby dalszej ewolucji narzędzia konieczne było złamanie tego wymagania i dokonanie głębokiej przebudowy *frameworku* tak, aby jeden plik zawierający projekt dla sterownika programowalnego zawierał w sobie pojedynczy test. Historycznie pojedynczy plik projektu mógł zawierać dziesiątki testów (od 2 do 70+), co prowadziło do nadmiernego ich rozrostu i braku stabilności, np. ze względu na występujące zależności od kolejności uruchamiania skryptów i trudności w selektywnym uruchamianiu ich fragmentów. Przy okazji wprowadzony został nowy system rejestrowania danych, w którym każdy krok testu generuje informację zawierającą wynik weryfikacji, opis tekstowy, a w razie potrzeby także wartości oczekiwane i rzeczywiste. Pozwoliło to przyspieszyć diagnozę problemów, ułatwiło identyfikację usterek oraz dało podstawy do późniejszej analizy przyczyn występowania defektów.

Zaletą tego rozwiązania była przede wszystkim większa stabilność i niezawodność, bardziej szczegółowe informacje zwracane przez skrypt po zakończeniu testu i niższy koszt utrzymania w przypadku pojedynczego testu. Idący za zmianami wzrost liczby plików (jeden test na projekt) wygenerował z kolei nowe wymagania narzędziowe poprzez konieczność zarządzania wersjami i przygotowanie odpowiednich szkieletów plików.

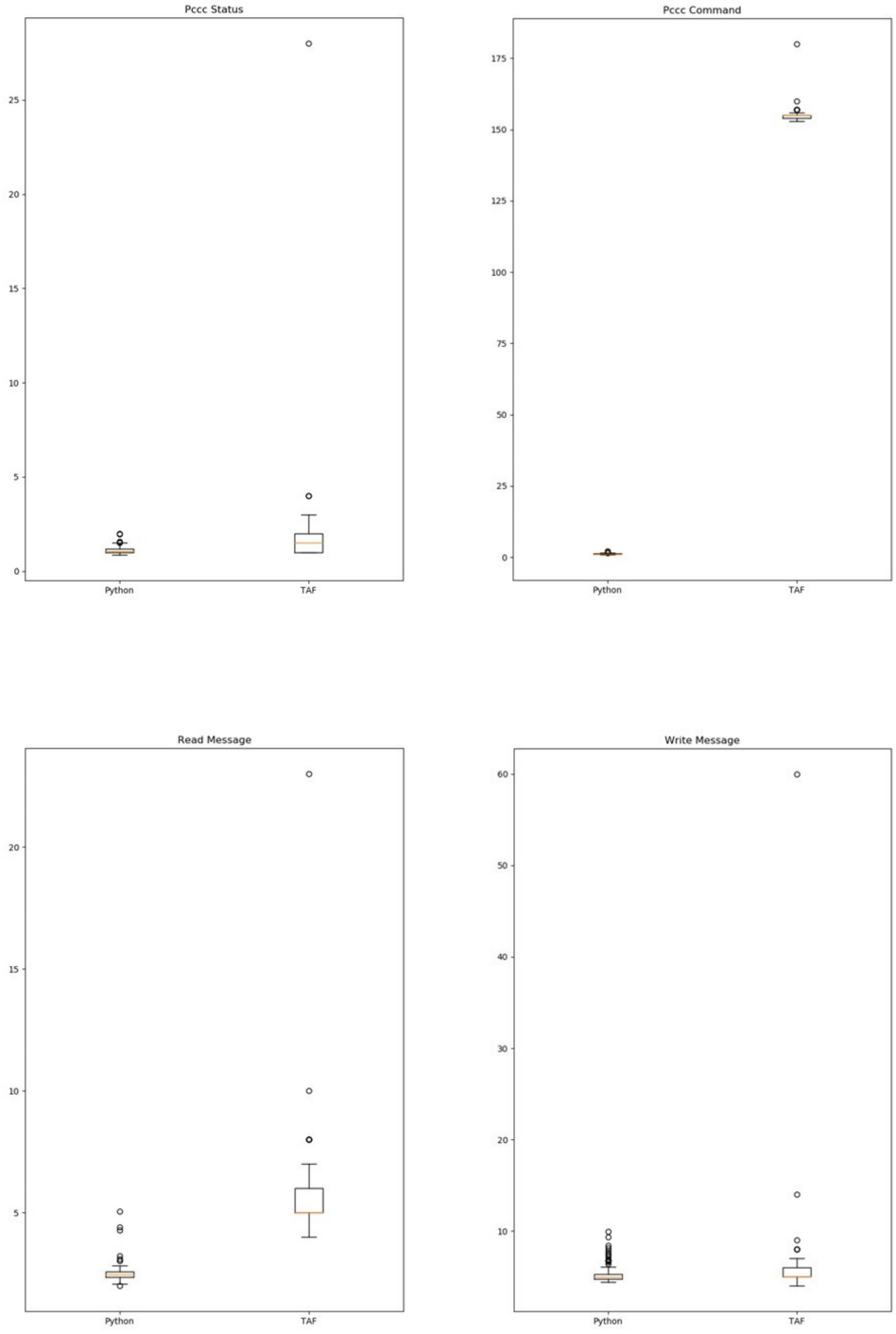
Zastosowanie w projektach Oscar i Lima nowego protokołu komunikacyjnego oraz struktury parametrów urządzeń wymusiło fundamentalne zmiany w podejściu do automatyzacji testów. Przede wszystkim dużym wyzwaniem było utrzymywanie własnego środowiska do automatyzacji testów, co wiązało się z kosztami i dużym nakładem pracy. Mimo oparcia się w dużym stopniu na języku graficznym, przystępność w obsłudze nie szła w parze z szybkością tworzenia skryptów testowych, czy też dostępnością

mechanizmów znanych ze współczesnych środowisk programistycznych, takich jak autouzupełnianie, czy pomoc kontekstowa. Występujące niejednokrotnie błędy w kodzie elementów odpowiedzialnych za poszczególne kroki nie były łatwe w naprawie, gdyż wymagały dostępu do plików źródłowych i osobnych licencji. Finalnie, w przypadku zatrudniania inżynierów testów nie można było liczyć na znajomość stosowanego *frameworku*, a nauczenie się jego obsługi, choć wymagane w pracy, nie zwiększało konkurencyjności pracownika w dalszej perspektywie rozwoju kariery zawodowej.

Aby zrealizować nowe projekty konieczne było więc zastosowanie innej technologii. Jednym z powszechnie używanych standardów jeśli chodzi o automatyzację testów jest korzystanie z języka Python, który oferuje wsparcie w postaci dedykowanych bibliotek (np. unittest, PyTest). Wyniki wstępnych eksperymentów mających na celu określenie deterministyczności nowego narzędzia prezentuje Rysunek 5.1, na którym widać wyrażony w milisekundach czas realizacji komend wysyłanych za pomocą różnych protokołów do systemu wbudowanego.

Na korzyść tego rozwiązania przemawiał fakt istnienia także dostępnej wewnętrznie biblioteki komunikacyjnej obsługującej protokoły komunikacyjne zgodne z projektami Oscar i Lima oraz ich strukturę parametrów. To oznaczało, że stworzenie nowego *frameworku* wiązałoby się z mniejszymi niż początkowo zakładano nakładami pracy. Automatyzacja testów w języku Python mogłaby być powszechnie wykorzystywana dla przyszłych projektów wspierających zbliżoną konstrukcję parametrów nawet, gdyby protokół komunikacyjny się różnił — w takim przypadku konieczna byłaby zmiana tylko biblioteki komunikacyjnej. Ponadto system mógłby służyć do wywoływania procedur stworzonych w programie Test Automation Framework lub programów dla sterownika programowalnego zawierających testy, czyli wspierać uruchamianie poprzednich rozwiązań.

Rozważanym krokiem było także wykorzystanie do automatyzacji testów dużych modeli językowych, które mogłyby generować scenariusze testowe oraz skrypty testów na podstawie specyfikacji wymagań funkcjonalnych. W tym celu w ramach prac wdrożeniowych dokonano próby *fine-tuningu* modelu OpenAI GPT 3.5 Turbo, do której wykorzystano prawie 800 par wymaganie–test, a następnie poproszono model o wygenerowanie testów na podstawie nieznanych mu wymagań o zbliżonym formacie. Pierwszym krokiem było dopasowanie danych wejściowych — zarówno wymagania, jak i istniejące przypadki testowe przygotowane były w formacie CSV, który wymagał ich konwersji do formatu czatu zrozumiałego dla modelu wraz z określeniem roli modelu (syste-



Rysunek 5.1: Porównanie rozkładu czasów realizacji komend

m/user/assistant), w tym wypadku do pliku JSONL, którego przykładową strukturę przedstawia Listing 5.1.

```
{"messages":[
  {"role":"system","content":"Jestes␣asystentem␣testera.␣Z␣
     ↪ wejsciowego␣wymagania␣generujesz␣kompletne␣przypadki␣testowe␣
     ↪ w␣formacie␣JSON␣zgodnym␣ze␣schematem␣organizacyjnym.␣Zawsze␣
     ↪ dolaczasz␣sledzenie␣do␣wymagan␣i␣stosujesz␣poziom␣
     ↪ szczegolowosci␣adekwatny␣do␣priorytetu."},
  {"role":"user","content":"[FR-1234]␣<Wymaganie>"},
  {"role":"assistant","content":"{␣\"requirement_id\":␣\"FR-1234\",␣
     ↪ \"test_cases\":␣[␣{␣\"id\":␣\"TC-FR-1234-01\",␣\"title\":␣
     ↪ \"<Tytul␣TC>\",␣\"preconditions\":␣[\"<Warunki␣wstepne>\"],␣
     ↪ \"steps\":␣[\"<Kroki>\"],␣\"expected_results\":␣
     ↪ [\"<Oczekiwane␣wyniki>\"],␣\"priority\":␣\"<Priorytet>\",␣
     ↪ \"trace_to_requirement\":␣\"FR-1234\"␣}␣]␣}"}
]}
```

Listing 5.1: Przykładowy fragment pliku JSONL w formacie chatu

Fragment JSONL przedstawia zapis konwersacji w formacie, który umożliwia przechowywanie sekwencji obiektów JSON w jednej strukturze, zrozumiałej przez model GPT 3.5 Turbo. W tym przypadku jeden obiekt zawierający klucz *messages* jest tablicą trzech komunikatów reprezentujących przebieg interakcji w stylu czatu. Każdy komunikat posiada atrybut *role*, określający nadawcę, oraz *content*, zawierający treść wiadomości. Pierwszy komunikat, z rolą *system*, definiuje kontekst pracy asystenta. Określa, że asystent pełni funkcję testera i na podstawie wejściowego wymagania ma wygenerować kompletne przypadki testowe w formacie JSON zgodnym ze schematem organizacyjnym. W treści instrukcji podkreślono konieczność zachowania śledzenia do wymagań oraz dostosowania poziomu szczegółowości do priorytetu. Drugi komunikat, z rolą *user*, zawiera wymaganie funkcjonalne oznaczone przykładowym identyfikatorem FR-1234. W tym miejscu kodu znajduje się element zastępczy, który w rzeczywistej sytuacji byłby zastąpiony opisem wymagania. Trzeci komunikat, z rolą *assistant*, przedstawia odpowiedź w formacie JSON. Odpowiedź zawiera klucz *requirement_id* z wartością FR-1234 oraz tablicę *test_cases*, w której znajduje się jeden przypadek testowy. Ten przypadek ma unikalny identyfikator, tytuł, warunki wstępne, kroki testowe, oczekiwane wyniki, priorytet oraz pole *trace_to_requirement*, które zapewnia powiązanie z wymaganiem. Dzięki temu zapewniona jest spójność dokumentacji testowej oraz możliwość łatwego śledzenia powiązań między wymaganiami a testami.

Przykładowa procedura scalenia wymagań i testów służących do treningu i walidacji modelu może wyglądać jak na Listingu 5.2. Skrypt w Pythonie realizuje kompletny, powtarzalny przepływ łączenia wymagań z odpowiadającymi im scenariuszami testowymi i przygotowuje trzy zbiory wyjściowe: dwa w formacie JSONL do treningu i walidacji modelu konwersacyjnego (symulującego interakcję *system–user–assistant*) oraz jeden w formacie CSV do późniejszej ewaluacji. Całość obejmuje wczytanie danych źródłowych, normalizację pól scenariusza, agregację testów per wymaganie, złączenie z tabelą wymagań, stratyfikowany podział na zbiory oraz serializację do docelowych formatów.

```
import pandas as pd, json, re
from pathlib import Path

REQ_CSV = "requirements.csv"
SCN_CSV = "scenarios.csv"
OUT_TRAIN = "train.jsonl"
OUT_VAL = "val.jsonl"
OUT_TEST = "test.csv"  # do pozniejszej ewaluacji

# 1) wczytaj
req = pd.read_csv(REQ_CSV)
scn = pd.read_csv(SCN_CSV)

# 2) scal scenariusze do jednego pola JSON na wymaganie
def split_list(s):
    if pd.isna(s) or not isinstance(s, str): return []
    return [x.strip() for x in s.split("|||") if x.strip()]

scn["Preconditions"] = scn["Preconditions"].apply(split_list)
scn["Steps"] = scn["Steps"].apply(split_list)
scn["Expected_Results"] = scn["Expected_Results"].apply(split_list)

cases = (
    scn.groupby("Requirement_ID")
       .apply(lambda g: [
           {
             "id": r["Test_Case_ID"],
             "title": r["Title"],
             "preconditions": r["Preconditions"],
             "steps": r["Steps"],
             "expected_results": r["Expected_Results"],
             "priority": r.get("Priority", "Medium"),
             "trace_to_requirement": r["Requirement_ID"]
```

```
            } for _, r in g.iterrows()
        ])
        .rename("test_cases")
        .reset_index()
)

ds = req.merge(cases, on="Requirement_ID", how="inner")

# 3) podzial: train/val/test (stratyfikacja po Priority)
from sklearn.model_selection import train_test_split
train, temp = train_test_split(ds, test_size=0.3,
    ↪ stratify=ds["Priority"], random_state=42)
val, test = train_test_split(temp, test_size=0.5,
    ↪ stratify=temp["Priority"], random_state=42)

def make_jsonl(df, out_path):
    with open(out_path, "w", encoding="utf-8") as f:
        for _, r in df.iterrows():
            system = ("Jestes␣asystentem␣QA.␣Generujesz␣kompletne␣
                ↪ przypadki␣testowe␣w␣JSON␣"
                      "zgodne␣z␣naszym␣schematem.␣Kazdy␣przypadek␣
                          ↪ zawiera␣preconditions,␣steps,␣"
                      "expected_results,␣priority␣oraz␣
                          ↪ trace_to_requirement.")
            user = f"[{r.Requirement_ID}]␣{r.Requirement_Text}"
            assistant = json.dumps({
                "requirement_id": r.Requirement_ID,
                "requirement_text": r.Requirement_Text,
                "test_cases": r.test_cases
            }, ensure_ascii=False)
            record = {"messages":[
                {"role":"system","content":system},
                {"role":"user","content":user},
                {"role":"assistant","content":assistant}
            ]}
            f.write(json.dumps(record, ensure_ascii=False) + "\n")

make_jsonl(train, OUT_TRAIN)
make_jsonl(val, OUT_VAL)
test.to_csv(OUT_TEST, index=False)
print("Gotowe:", OUT_TRAIN, OUT_VAL, OUT_TEST)
```

Listing 5.2: Przykładowy skrypt do scalenia wymagań i testów

Kolejnym krokiem w celu wykonania dostrojenia modelu była konfiguracja środowiska Azure OpenAI oraz rozpoczęcie zadania. Postęp zadania może być monitorowany, a następnie wdrożenie powinno zostać wywołane jako nowy model, co widoczne jest na Listingu 5.3. Kod przedstawia kompletny proces *fine-tuningu* modelu Azure OpenAI: od konfiguracji klienta API z wykorzystaniem zmiennych środowiskowych, przez przesłanie plików JSONL z danymi treningowymi i walidacyjnymi, aż po utworzenie zadania dostrajania modelu GPT 3.5 Turbo. Po zakończeniu treningu skrypt umożliwia monitorowanie statusu i pobranie nazwy wytrenowanego modelu, a następnie wykorzystanie go w rozmowie poprzez wywołanie wdrożenia z przygotowaną strukturą wiadomości. Dzięki temu możliwe jest uzyskanie spersonalizowanego modelu zoptymalizowanego pod kątem automatyzacji tworzenia przypadków testowych.

```
import os
from openai import AzureOpenAI

# 1) Konfiguracja API Azure OpenAI
client = AzureOpenAI(
    api_key=os.environ["AZURE_OPENAI_API_KEY"],
    azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT"],  # adres
        ↪ endpointu
    api_version="2023-05-15"                              # wersja API
)
# 2) Upload plikow JSONL
train = client.files.create(file=open("train.jsonl","rb"),
    ↪ purpose="fine-tune")
val   = client.files.create(file=open("val.jsonl","rb"),
    ↪ purpose="fine-tune")

# 3) Utworz job fine-tuning (SFT)
job = client.fine_tuning.jobs.create(
    model="gpt-35-turbo",            # wybor modelu
    training_file=train.id,
    validation_file=val.id,
    hyperparameters={"n_epochs": 3, "batch_size":"auto",
        ↪ "learning_rate_multiplier":"auto"}
)
print("JOB:", job.id)

# 4) Monitorowanie postepu
job = client.fine_tuning.jobs.retrieve(job.id)
print(job.status, job.fine_tuned_model)
```

```
26
27 # 5) Wywolanie wdrozenia
28 deployment = "tc-gen-gpt35-ft"  # nazwa Deploymentu customowego modelu
29
30 messages = [
31   {"role":"system","content":"Jestes asystentem testera... Wygeneruj
       ↪ JSON zgodny z naszym schematem."},
32   {"role":"user","content":"<Tekst nowego wymagania>"}
33 ]
34
35 resp = client.chat.completions.create(
36     model=deployment,
37     messages=messages,
38     temperature=0.2
39 )
40
41 answer = resp.choices[0].message.content
42 print(answer)
```

Listing 5.3: Przykładowy skrypt do scalenia wymagań i testów

Aby uzyskać rzetelne wyniki, typowo wymagana jest co najmniej dwucyfrowa liczba przykładów. Niestety w odniesieniu do generowania testów na podstawie wymagań, występowały problemy ze spójnością i formatowaniem, a także spora różnorodność danych wejściowych. Powodowało to znaczące halucynacje modelu i brak powtarzalności odpowiedzi, szczególnie w kontekście stosowania precyzyjnych nazw parametrów urządzenia, czy narzędzi testerskich. W takiej sytuacji alternatywnym podejściem mogłoby być wykorzystanie techniki generowania wspomaganego wyszukiwaniem (ang. Retrieval-Augemented Generation, RAG). Technika ta zamiast trenowania modelu pozwala dynamicznie wstrzykiwać aktualne informacje (np. wymagania) w momencie generowania odpowiedzi. Historyczne przypadki testowe i wytyczne projektowe mogłyby zostać zindeksowane w wektorowej bazie, a dokumenty wejściowe opatrzone metadanymi zawierającymi numer identyfikacyjny wymagania, priorytet, czy też oczekiwany rodzaj testu. Na podstawie nowego wymagania, model wyszukałby w bazie danych najbardziej podobne istniejące scenariusze testowe, traktował je jako kontekst i na tej podstawie generował nowe przypadki testowe. Jeszcze innym podejściem mogłoby być wykorzystanie uczenia przez wzmocnienie (ang. Reinforcement Learning, RL), w którym odpowiedź modelu każdorazowo jest oceniana na podstawie informacji zwrotnej lub automatycznych metryk. W tym wariancie model generuje kilka wariantów testów

dla tego samego wymagania, które następnie zostają poddane ocenie — najlepsze odpowiedzi są nagradzane, a model uczy się maksymalizować szansę na jej otrzymanie. Wszystkie te metody mogą być implementowane współbieżnie w celu otrzymania lepszych rezultatów.

5.2 *Framework* testowy

Istnieje wiele typów architektury *frameworków* testowych, która może być zastosowana przy tworzeniu nowego narzędzia. *Frameworki* liniowe, znane również jako „nagraj i odtwórz" sprawdzają się w odniesieniu do prostych przypadków testowych, w których sekwencyjnie uruchamiane skrypty odpowiedzialne są za testowanie systemu. Nie wymagają zaawansowanej wiedzy ani pisania kodu, są łatwe do zrozumienia, szybkie do wdrożenia i łatwo integrowalne z istniejącymi procesami. Przykładem takiego rozwiązania jest *framework*, który stosowany był do automatyzacji testów przed projektami Oscar i Lima. Istotną wadą takiego rozwiązania jest mała elastyczność w zakresie reużywalności testów i obsługi różnych zestawów danych wejściowych, a także wysokie koszty utrzymywania testów ze względu na konieczność częstej ich przebudowy. W podejściu modułowym, testy podzielone są na mniejsze jednostki (moduły), które można testować niezależnie, a w razie potrzeby połączyć w większe scenariusze. Zaletą takiej architektury jest większa elastyczność i możliwość szybkiej modyfikacji, wysoka reużywalność i skalowalność testów. Niestety pociąga to za sobą bardziej złożone wdrożenie, które wymaga umiejętności programistycznych.

Podejście modułowe może być rozszerzone o zidentyfikowanie powtarzalnych funkcji i pogrupowaniu ich w biblioteki, które można wykorzystywać w różnych testach. To sprawia, że raz napisany kod może być wywoływany w różnych miejscach, co przynosi duże oszczędności w zakresie nakładu pracy. Rośnie natomiast sama złożoność i wymagania techniczne, aby właściwie zidentyfikować wspólne funkcje. Dobrą praktyką jest również oddzielenie danych wejściowych od logiki testów. Jest to architektura szczególnie opłacalna, gdy istnieje konieczność testowania tych samych funkcjonalności z różnymi zestawami danych. Rozwiązanie to można dalej rozwijać o rozpoznawanie słów kluczowych, na przykład w testowaniu graficznych interfejsów użytkownika (ang. Graphic User Interface, GUI), jeśli wymaga tego dany produkt.

W praktyce najczęściej stosuje się architektury hybrydowe, to jest łączące cechy różnych *frameworków*, aby dopasować się do potrzeb danego środowiska. Pozwala to wyko-

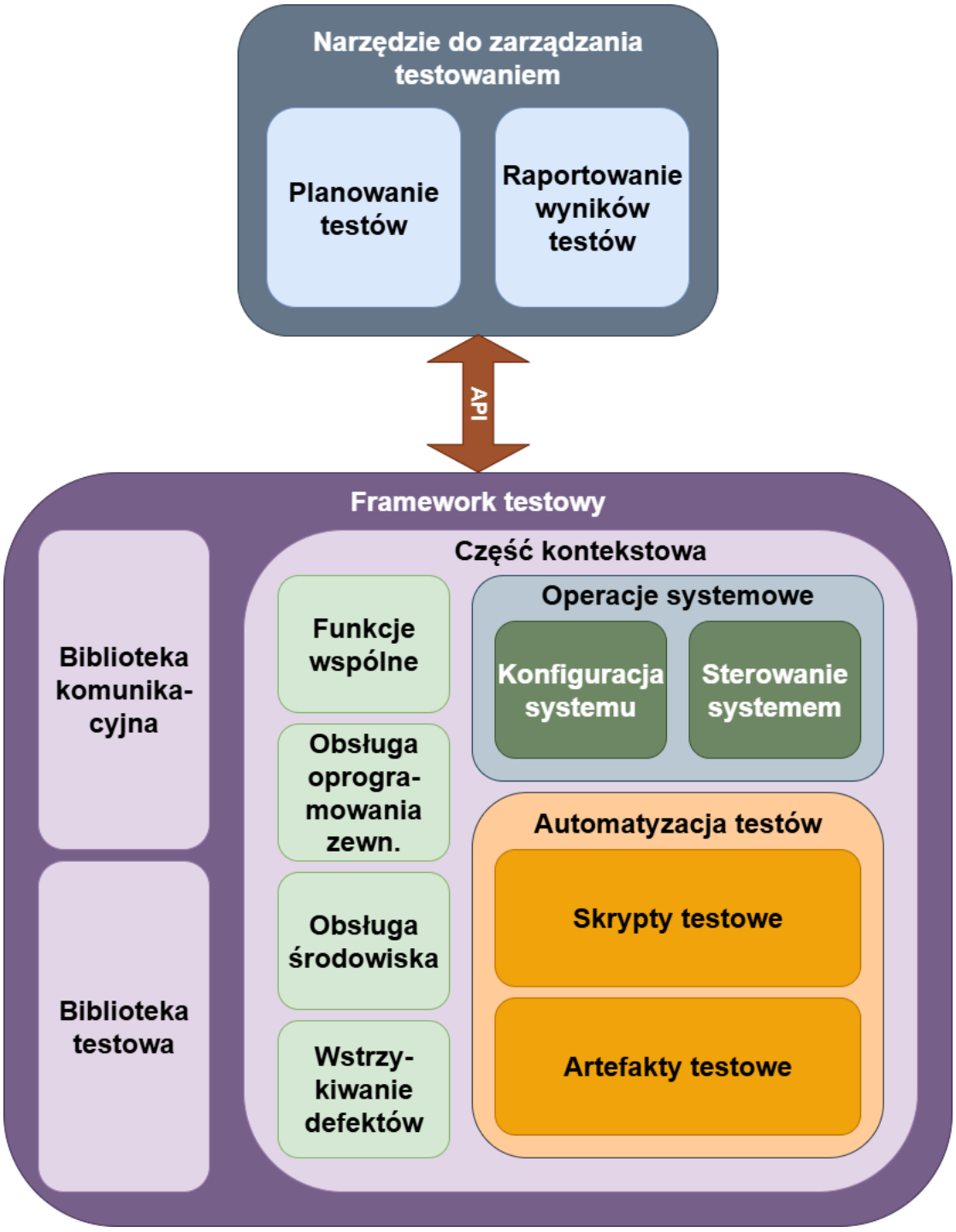
rzystać mocne strony różnych podejść i zminimalizować ich wady. Wdrożony *framework* testowy[2] jest rozwiązaniem modułowym, opartym o wspólne biblioteki i oddzielającym dane wejściowe od skryptów, spełnia więc kryteria *frameworku* hybrydowego. Składa się z kombinacji narzędzi i praktyk, które ułatwiają proces testowania oraz automatyzacji testów. Dzięki standaryzacji zapewnia szybkość i prostotę w programowaniu skryptów oraz spójność w analizie i raportowaniu wyników. Obejmuje on:

- standardy kodowania i dokumentację,
- obsługę danych testowych,
- organizację repozytorium,
- warstwę komunikacyjną,
- obsługę i interakcję z testowanym systemem,
- konfigurację *logów* testowych,
- zasady pisania przypadków testowych,
- wywoływanie skryptów testowych,
- praktyki dotyczące raportowania wyników.

Architektura *frameworku* przedstawiona została na Rysunku 5.2. Obejmuje ona również wymianę informacji z narzędziem do zarządzania testowaniem, z którym wymiana informacji odbywa się poprzez dedykowany interfejs programowania aplikacji (ang. Application Programming Interface, API), udostępniony wraz z narzędziem przez dostawcę. Planowanie testów, ich uruchamianie oraz raportowanie wyników obsługiwane są przez tę platformę, podczas gdy pozostała część operacji wykonywana jest poprzez *framework*.

Głównymi elementami *frameworku* są biblioteki: komunikacyjna i testowa, oraz część kontekstowa, to jest obszar bezpośrednio związany ze specyfiką testowanych systemów wbudowanych. Można uznać, że obie biblioteki stanowią rozwiązanie uniwersalne, natomiast część kontekstowa mocno zależy od indywidualnej charakterystyki projektów. Aspekt komunikacji dotyczy bezpośrednio nawiązywania połączenia z testowanym

[2]Ze względu na poziom złożoności, praca nad implementacją i rozwojem *frameworku* była prowadzona przez zespół inżynierów testów. Wkładem autora jest sformułowanie wymagań, a także kod części modułów i testów.



Rysunek 5.2: Architektura *frameworku* testowego

urządzeniem. Może odbywać się to za pomocą sieci komputerowej (np. Ethernet/IP), lub innego protokołu komunikacyjnego. Wydzielenie tej części na zewnątrz w formie biblioteki powoduje, że w przyszłości można swobodnie zmienić sposób połączenia z te-

stowanym systemem, nie modyfikując przy tym części kontekstowej. W omawianym wdrożeniu za komunikację odpowiadała wewnętrznie rozwijana biblioteka napisana w języku Python.

Przez bibliotekę testową rozumiany jest pakiet dostarczający kompletne rozwiązanie w zakresie obsługi testowania, w skład którego wchodzą m.in. silnik testowy, klasy reprezentujące zbiory testów, konfiguratory, czy opcje parametryzacji testów. Przykładem takich rozwiązań dla języka Python są narzędzia Unittest i Pytest. Moduł Unittest w Pythonie inspirowany jest *frameworkiem* JUnit (napisany w Javie) i od lat wchodzi w skład biblioteki standardowej, dzięki czemu nie wymaga dodatkowej instalacji i jest odbierany jako bezpieczny, przewidywalny wybór w projektach. Pytest pozostaje natomiast jednym z najpopularniejszych, otwarto-źródłowych (ang. open-source) *frameworków* testowych, cenionym za szybkość pracy i ekosystem rozszerzeń.

Z perspektywy filozofii i ergonomii różnice wynikają głównie ze sposobu organizowania testów i zarządzania ich cyklem życia. Unittest wykorzystuje klasy *TestCase* z metodami testowymi oraz przewidywalny cykl konfiguracji środowiska, a następnie jego przywracania do stanu pierwotnego (ang. setUpClass/tearDownClass). Wyposażony jest także w bogaty zestaw metod asercji, co dobrze wpisuje się w środowiska oczekujące silnie ustrukturyzowanych testów. Pytest kładzie nacisk na prostotę, gdzie testy pisane są jako funkcje lub metody i dzielone na zakresy (ang. scopes), a konfiguracja i czyszczenie środowiska realizowane jest specjalną funkcją, tzw. *fixture*.

W ramach części kontekstowej można wyróżnić moduł funkcji wspólnych, to jest reużywalnych fragmentów kodu, które mogą być następnie wywoływane w skryptach testowych. Ułatwia to ich utrzymywanie, gdy dochodzi do znaczącej zmiany w konfiguracji urządzenia, a także skraca czas potrzebny na pisanie testów automatycznych. Funkcje wspólne mocno zależą od potrzeb projektowych, ale mogą należeć do nich na przykład: nadpisanie numeru katalogowego urządzenia, odczyt lub modyfikacja danych diagnostycznych, tworzenie wykresów w raportach, czy też wymuszenie ponownego uruchomienia systemu.

Moduły obsługi oprogramowania zewnętrznego i środowiska odpowiadają za elementy zewnętrzne, takie jak współpraca z urządzeniami pomiarowymi (np. oscyloskopami, analizatorami widma), symulatorami, systemami testowania sprzętu lub oprogramowania w pętli (HiL lub SiL), czy elementami wyposażenia stanowisk testowych (na przykład kontrola przełączników sieciowych). Wywołują też potrzebne oprogramowanie nie będące częścią *frameworku*, a mogące wchodzić w interakcje z testowanym

systemem.

Kolejnym istotnym elementem jest moduł wstrzykiwania defektów, który jest w stanie wymusić wywołanie określonych warunków skutkujących zgłoszeniem przez testowany system określonej usterki w formie ostrzeżenia lub błędu. Może odbywać się to w pełni programowo, jeśli testowany system udostępnia taką funkcjonalność w środowisku inżynierskim, albo sprzętowo z wykorzystaniem *debuggera* albo systemu Hardware-in-the-Loop. Rozwiązanie to umożliwia nadpisanie odczytywanych przez czujniki systemu wbudowanego parametrów diagnostycznych, takich jak na przykład temperatura, napięcie, częstotliwość. Innym zastosowaniem może być też przyspieszenie liczników diagnostycznych w systemach obsługujących funkcjonalności związane z konserwacją predykcyjną (ang. predictive maintenance).

Komponent operacji systemowych odpowiada za obsługę bazy danych parametrów konfiguracyjnych systemu oraz tłumaczenie instrukcji i kroków skryptów automatycznych, napisanych w Pythonie w formie dopasowanej do potrzeb człowieka (zbliżonej do języka naturalnego), na język zrozumiały dla testowanego systemu wbudowanego, czerpiąc przy tym w dużej mierze z biblioteki komunikacyjnej. Umożliwia przeprowadzenie i zmianę konfiguracji systemu oraz wysyłanie instrukcji sterujących, które wymuszają wykonanie przez system określonych operacji. W odniesieniu do urządzeń automatyki przemysłowej może to być na przykład zmiana zadanej częstotliwości napędu i uruchomienie sterowania silnikiem indukcyjnym.

Wszystkie te składniki są konieczne, aby przystąpić do właściwej automatyzacji testów, to jest programowania skryptów, obsługi danych testowych, a także generowania *logów* i raportów z wykonanych testów. Przykład zautomatyzowanego testu dla systemu wbudowanego przedstawia Listing 5.4. Zawiera on wywołanie metod odpowiedzialnych za przygotowanie środowiska do testu oraz przywrócenie stanu pierwotnego po jego zakończeniu, a także kilka przypadków testowych dla przykładowej klasy *LedController*. Struktura klasy TestLedController obejmuje metody cyklu życia testów: *setUp()* inicjalizuje nowy obiekt kontrolera przed każdym testem, a *tearDown()* wykonuje czynności porządkowe po zakończeniu testu, co zapewnia izolację przypadków i brak efektów ubocznych. Testy sprawdzają cztery kluczowe scenariusze: stan początkowy, włączenie, wyłączenie oraz przełączanie. Każdy test korzysta z asercji, aby potwierdzić zgodność rzeczywistego stanu LED z oczekiwanym zachowaniem po wywołaniu odpowiednich metod. Dzięki temu kod zapewnia pełne pokrycie podstawowej logiki kontrolera, weryfikując zarówno inicjalizację, jak i reakcję na operacje sterujące. Na potrzeby pisania

testów automatycznych przyjęty został też nieformalny standard kodowania, zaproponowany przez społeczność rozwijającą język Python w formie rozszerzenia specyfikacji (ang. Python Enhancement Proposals, PEP), który opisany jest w dokumentach *PEP 8 — Style Guide for Python Code* oraz *PEP 20 — The Zen of Python*.

```
1  # test_led_controller.py
2  import unittest
3  from led_controller import LedController
4
5  class TestLedController(unittest.TestCase):
6
7      def setUp(self):
8          """Przygotowanie srodowiska testowego przed kazdym testem."""
9          self.led = LedController()
10         print(">>> setUp: Tworzenie nowego obiektu LedController")
11
12     def tearDown(self):
13         """Czyszczenie po kazdym tescie."""
14         print(">>> tearDown: Test zakonczony")
15
16     def test_initial_state(self):
17         """Sprawdzenie stanu poczatkowego LED."""
18         self.assertFalse(self.led.is_on(), "LED powinna byc wylaczona
               ↪ na starcie")
19
20     def test_turn_on(self):
21         """Sprawdzenie wlaczenia LED."""
22         self.led.turn_on()
23         self.assertTrue(self.led.is_on(), "LED powinna byc wlaczona
               ↪ po turn_on()")
24
25     def test_turn_off(self):
26         """Sprawdzenie wylaczenia LED."""
27         self.led.turn_on()
28         self.led.turn_off()
29         self.assertFalse(self.led.is_on(), "LED powinna byc wylaczona
               ↪ po turn_off()")
30
31     def test_toggle(self):
32         """Sprawdzenie przelaczania LED."""
33         self.led.toggle()
34         self.assertTrue(self.led.is_on(), "LED powinna byc wlaczona
```

```
            ↪ po␣pierwszym␣toggle()")
35          self.led.toggle()
36          self.assertFalse(self.led.is_on(), "LED␣powinna␣byc␣wylaczona␣
                ↪ po␣drugim␣toggle()")
37
38  if __name__ == '__main__':
39      unittest.main()
```

Listing 5.4: Przykładowy skrypt testu automatycznego

Repozytorium które przechowuje kod *frameworku*, testy automatyczne, dane testowe i pozostałe artefakty oferuje możliwość uruchomienia systemu ciągłej integracji, w którym określone testy uruchamiane są każdorazowo po dostarczeniu przez programistów nowej wersji oprogramowania. Grupa tych testów została wyznaczona w taki sposób, aby obejmować podstawowe funkcjonalności systemu i dzięki temu zminimalizować ryzyko, że dana wersja oprogramowania zawiera błędy, które w oczywisty sposób blokują lub uniemożliwiają dalsze wyspecjalizowane testowanie.

Ważnym aspektem związanym z *frameworkiem*, jest również jego dokumentacja. W opisywanym rozwiązaniu wykorzystano podejście zwane kodem samodokumentującym się (ang. self-documented code) z wykorzystaniem generatora dokumentacji Sphinx. Samodokumentujący kod to taki, który dzięki swojej strukturze, nazwom i komentarzom jest zrozumiały bez dodatkowych wyjaśnień. W Pythonie kluczową rolę dla spełnienia tych kryteriów odgrywają *docstringi* — komentarze umieszczane w klasach, funkcjach i modułach. Dzięki nim kod staje się bardziej czytelny i łatwiejszy w utrzymaniu. *Docstringi* opisują działanie elementów kodu, parametry, typy zwracanych wartości i ewentualne wyjątki. To fundament, na którym Sphinx buduje automatyczną dokumentację, przekształcając je na format HTML, PDF lub inny. Dzięki temu dokumentacja jest spójna, estetyczna i łatwa do nawigacji, szczególnie gdy projekt podąża za rozszerzeniem specyfikacji *PEP 257 — Docstring Conventions*.

5.3 Ustanowienie ram dla zakresu testów

Ramy postępowania w testach złożonych systemów wbudowanych zostały wyznaczone na podstawie przeprowadzonych badań oraz wdrożeń w projektach Oscar i Lima. Celem ich opracowania było stworzenie uniwersalnego, a zarazem praktycznego mechanizmu, który umożliwi optymalizację procesu testowania poprzez świadome zarządza-

nie zakresem, priorytetami oraz formą testów. Ramy te integrują podejście oparte na analizie ryzyka, atomizację przypadków testowych, systematyczne wykorzystanie testowania eksploracyjnego oraz deterministyczną automatyzację, opartą o hybrydowy *framework* testowy zbudowany w języku Python i zintegrowany z narzędziem do zarządzania testami oraz środowiskiem ciągłej integracji.

Podstawowym założeniem przyjętym w ramach procesu wyznaczania zakresu testów jest kierowanie się oceną ryzyka, a nie rutyną czy historycznymi schematami. Każdy scenariusz testowy podlega ocenie według macierzy ryzyka, gdzie ryzyko definiowane jest jako iloczyn prawdopodobieństwa wystąpienia defektu oraz potencjalnych konsekwencji jego materializacji. Takie podejście pozwala na racjonalne gospodarowanie zasobami oraz skupienie uwagi zespołu testerskiego na obszarach o największym wpływie na jakość i bezpieczeństwo produktu. W praktyce wdrożonej w projektach Oscar i Lima, ocena ryzyka stanowiła kluczowy czynnik decydujący o formie testu, jego częstotliwości oraz kwalifikacji do puli regresyjnej.

Kolejnym filarem ram jest atomizacja przypadków testowych, polegająca na powiązaniu każdego testu z pojedynczym wymaganiem funkcjonalnym. Takie podejście umożliwia precyzyjne śledzenie powiązań pomiędzy wymaganiami a testami, ułatwia analizę wpływu zmian oraz pozwala na szybkie określenie zakresu testów wymagających powtórzenia po modyfikacji oprogramowania. W projektach Oscar i Lima udało się osiągnąć atomizację na poziomie 99 % przypadków testowych, co znacząco usprawniło proces zarządzania testami oraz analizę pokrycia wymagań.

W ramach wdrożonych rozwiązań szczególną rolę odgrywa warstwowa regresja testów, podzielona na trzy poziomy: codzienną, jednokrotną oraz rozszerzoną. Przypisanie testów do odpowiedniej puli regresyjnej odbywa się na podstawie oceny ryzyka oraz kontekstu biznesowego. Testy o najwyższym ryzyku oraz kluczowym znaczeniu dla bezpieczeństwa i stabilności produktu kwalifikowane są do codziennej regresji, realizowanej w środowisku ciągłej integracji. Testy o średnim ryzyku oraz te, które powstały w wyniku testowania eksploracyjnego i uzyskały odpowiednią deterministykę, włączane są do regresji jednokrotnej, wykonywanej w cyklu wydania produktu lub interwału planowania. Pozostałe testy, o niskim ryzyku lub specyficznym charakterze, realizowane są w ramach regresji rozszerzonej, uruchamianej w miarę dostępności zasobów lub w odpowiedzi na szczególne wymagania biznesowe.

Testowanie eksploracyjne, wdrożone jako usystematyzowana technika w projektach Oscar i Lima, stanowi istotne uzupełnienie tradycyjnych metod testowania. ET po-

zwala na wykrycie defektów trudnych do odtworzenia oraz luk w pokryciu testowym, szczególnie w obszarach o wysokiej złożoności interfejsów modułów oraz w sytuacjach braku pełnej specyfikacji wymagań. Wyniki testów eksploracyjnych są systematycznie przekształcane w testy skryptowe i włączane do puli regresyjnej, gdy uzasadnia to efektywność ekonomiczna i techniczna. W projekcie Oscar dziewięć sesji ET pozwoliło na wykrycie sześciu istotnych defektów, z których pięć nie byłoby możliwe do wykrycia przez testy automatyczne lub skryptowe ze względu na ich niską odtwarzalność.

Automatyzacja testów w ramach wdrożonego frameworku opiera się na rygorystycznych kryteriach stabilności i niezawodności. Do puli regresyjnej kwalifikowane są wyłącznie testy spełniające ustalone progi stabilności (np. 10/10, 25/25, 48/50, 95/100 powtórzeń bez wyników fałszywie pozytywnych lub negatywnych), co eliminuje szumy w sygnale testowym i zwiększa wiarygodność procesu decyzyjnego. Testy zależne od znanych defektów przechodzą w tryb ostrzeżenia, z odpowiednim powiązaniem do zgłoszenia w systemie zarządzania defektami, co pozwala na zachowanie przejrzystości raportowania i ułatwia późniejszą aktualizację.

Proces wyznaczania zakresu testów wspierany jest przez zestaw metryk i kryteriów wejścia/wyjścia, takich jak współczynnik efektywności testów (TER), gęstość defektów (FD), niezawodność automatyzacji (TAR), procent wykrytych defektów (DDP) oraz wykresy spalania (burndown chart). Metryki te służą do monitorowania postępu kampanii testowej, oceny skuteczności wdrożonych zmian oraz podejmowania decyzji o zakończeniu testów. Przykładowo, wejście do kampanii testowej wymaga przejścia przez zestaw testów promocyjnych w środowisku CI, braku blokujących defektów bezpieczeństwa oraz dostępności obiektów testowych, natomiast wyjście uzależnione jest od osiągnięcia założonego pokrycia wymagań, spełnienia progów metryk oraz braku defektów krytycznych.

Oprócz powyższych, mierzony jest również dzienny przyrost nowych przypadków testowych w projekcie, a także liczba wykonanych przypadków testowych (z podziałem na ich poszczególne rezultaty) i wykrytych defektów. Dane te pozwalają ocenić postęp w realizacji projektu, a także wykryć ewentualne czynniki blokujące. Kondycja projektu podlega także ocenie na podstawie liczby testów gotowych do uruchomienia, w porównaniu do testów dopiero w trakcie projektowania, wymagających aktualizacji, zbędnych, bądź wyłączonych ze względu na inne czynniki. Śledzony w sposób automatyczny jest także poziom pokrycia testami wymagań funkcjonalnych.

W efekcie wdrożenia opisanych ram, projekty Oscar i Lima osiągnęły znaczący

wzrost efektywności testowania, potwierdzony wysokimi wartościami TER i FD w porównaniu do projektów referencyjnych. Skrócono czas kampanii testowej dzięki automatyzacji i atomizacji przypadków testowych, a testowanie eksploracyjne pozwoliło na wykrycie defektów o wysokiej istotności dla klienta. Stabilność procesu została zapewniona przez rygorystyczne kryteria kwalifikacji testów do regresji oraz systematyczne monitorowanie metryk jakościowych.

Podsumowując, ustanowione ramy dla zakresu testów stanowią spójny, mierzalny i elastyczny mechanizm zarządzania procesem testowania funkcjonalnego złożonych systemów wbudowanych. Integracja podejścia opartego na ryzyku, atomizacji przypadków testowych, testowania eksploracyjnego oraz deterministycznej automatyzacji pozwala na optymalizację procesu testowego, zwiększenie jakości produktu oraz efektywne wykorzystanie zasobów zespołu testerskiego. Wyniki wdrożenia potwierdzają zasadność przyjętych rozwiązań i wskazują kierunek dalszego rozwoju praktyk testowania w przedsiębiorstwie.

Rozdział 6

Podsumowanie i wnioski

6.1 Najważniejsze osiągnięcia projektu doktorskiego

W ramach realizacji projektu doktorskiego osiągnięto szereg kluczowych rezultatów, które istotnie wpłynęły na proces testowania funkcjonalnego złożonych systemów wbudowanych w przedsiębiorstwie Rockwell Automation. Przeprowadzone wdrożenia oraz badania pozwoliły na wypracowanie nowoczesnych ram postępowania testowego, które zostały uznane za wytyczne procesowe i wdrożone jako standard w organizacji. Osiągnięcia te mają wymierny wpływ zarówno na efektywność testowania, jak i na jakość dostarczanych rozwiązań. Najważniejsze elementy projektu można podsumować w następujących punktach:

1. Znaczący wzrost efektywności testowania — potwierdzony wysokimi wartościami współczynników efektywności testów TER (ang. Test Effectiveness Ratio) i gęstości błędów FD (ang. Fault Density) dla projektów wzorcowych (Oscar, Lima) na tle grup referencyjnych, a także większa stabilność procesu dzięki rygorystycznym progom kwalifikacji do regresji i ciągłemu monitorowaniu metryk.

2. Wdrożenie hybrydowego *frameworku* testowego opartego na języku Python, zintegrowanego z narzędziami do zarządzania testami oraz środowiskiem ciągłej integracji. Narzędzie to umożliwiło efektywną automatyzację testów oraz szybkie dostosowanie narzędzi do specyfiki różnych projektów i wymagań.

3. Systematyczne wykorzystanie testowania eksploracyjnego (ET) — wdrożenie ET

jako usystematyzowanej techniki pozwoliło na wykrycie defektów trudnych do odtworzenia oraz luk w pokryciu testowym, szczególnie w obszarach o wysokiej złożoności interfejsów modułów oraz w sytuacjach braku pełnej specyfikacji wymagań. Wyniki testów eksploracyjnych były przekształcane w testy skryptowe i włączane do puli regresyjnej.

4. Atomizacja przypadków testowych — każdy przypadek testowy został powiązany z pojedynczym wymaganiem funkcjonalnym, co pozwoliło na precyzyjne śledzenie powiązań, łatwą analizę wpływu zmian oraz szybkie określenie zakresu testów wymagających powtórzenia po modyfikacji oprogramowania. W projektach Oscar i Lima osiągnięto atomizację na poziomie 99 %.

5. Integracja podejścia opartego na analizie ryzyka — ocena ryzyka stała się kluczowym czynnikiem decydującym o formie testu, jego częstotliwości oraz kwalifikacji do puli regresyjnej. Pozwoliło to na racjonalne gospodarowanie zasobami oraz skupienie uwagi zespołu testerskiego na obszarach o największym wpływie na jakość i bezpieczeństwo produktu.

6. Wprowadzenie rygorystycznych kryteriów stabilności i niezawodności testów automatycznych — do puli regresyjnej kwalifikowane były wyłącznie testy spełniające ustalone progi stabilności (np. 10/10, 25/25, 48/50, 95/100 powtórzeń bez wyników fałszywie pozytywnych lub negatywnych), co zwiększyło wiarygodność procesu.

Podsumowując realizację projektu doktorskiego, można jednoznacznie stwierdzić, że postawiona teza rozprawy została potwierdzona w toku przeprowadzonych badań i wdrożeń. Zastosowanie hybrydowego *frameworku* testowego, opartego na synergii deterministycznej automatyzacji, systematycznej analizy ryzyka, atomizacji przypadków testowych oraz usystematyzowanego testowania eksploracyjnego, umożliwiło mierzalną optymalizację procesu weryfikacji złożonych systemów wbudowanych. Potwierdzeniem tego są statystycznie istotne wzrosty wartości metryk jakościowych — w szczególności współczynnika efektywności testów (TER) oraz gęstości defektów (FD) — w projektach Oscar i Lima, które wyraźnie przewyższyły wyniki grup referencyjnych.

Cele badawcze zostały zrealizowane w pełnym zakresie. Po pierwsze, wdrożenie strategii testowania opartej na analizie ryzyka oraz automatyzacji testów przełożyło się na skrócenie czasu kampanii testowej i wzrost efektywności wykrywania defektów, co potwierdzają szczegółowe analizy porównawcze oraz zestawienia metryk w rozdziale

wyników. Po drugie, zastosowanie testowania eksploracyjnego jako integralnej części kampanii testowej pozwoliło na wykrycie defektów trudnych do odtworzenia, które nie zostałyby ujawnione w ramach testów formalnych — przykładem są scenariusze z projektu Oscar, gdzie testy eksploracyjne wykryły błędy o wysokiej istotności dla klienta. Po trzecie, wdrożenie nowych strategii testowania w zespołach rozproszonych zostało wsparte przez standaryzację procesów, automatyzację metryk oraz szkolenia, co umożliwiło skuteczną adaptację rozwiązań w różnych lokalizacjach i projektach.

Wypracowane autorskie koncepcje i rekomendacje wpisane w ramy postępowania testowego zostały uznane za wytyczne procesowe i wdrożone jako standard w przedsiębiorstwie, potwierdzając zasadność przyjętych rozwiązań oraz wskazując kierunek dalszego rozwoju praktyk testowania w sektorze systemów wbudowanych. Wdrożenie powyższych rozwiązań przyczyniło się do skrócenia czasu kampanii testowej, zwiększenia wykrywalności defektów oraz poprawy jakości dostarczanych produktów. Opracowane ramy postępowania testowego stanowią spójny, mierzalny i elastyczny mechanizm zarządzania procesem testowania funkcjonalnego złożonych systemów wbudowanych, integrując podejście oparte na ryzyku, atomizację przypadków testowych, testowanie eksploracyjne oraz deterministyczną automatyzację.

6.2 Implementacja opracowanej metody testowania w przedsiębiorstwie

W ramach realizacji pracy doktorskiej autor pełnił rolę inżyniera testów oraz kierownika projektu testowania funkcjonalnego złożonych systemów wbudowanych, odpowiadając za kompleksowe zarządzanie całym cyklem testowym. Do głównych zadań na tym stanowisku należało przygotowanie szczegółowego planu testów, w tym dobór i opracowanie strategii testowej dostosowanej do specyfiki projektów prowadzonych w przedsiębiorstwie Rockwell Automation. Autor nie tylko projektował i wykonywał istotną część testów w projektach Sierra, Quebec, Oscar i Lima, ale również był odpowiedzialny za sporządzenie końcowych raportów podsumowujących wyniki kampanii testowych oraz rekomendacje dotyczące dalszego rozwoju procesów jakościowych. Wkładem autora było również sformułowanie wymagań dla *frameworku* testowego, a także kod części modułów i testów automatycznych.

Decyzje o wdrożeniu poszczególnych elementów strategii testowej były podejmowa-

ne na podstawie autorskich pomysłów i rekomendacji autora, które — po akceptacji przez firmę — zostały włączone jako integralna część procesu testowego. W praktyce oznaczało to, że autor nie tylko inicjował zmiany w podejściu do testowania, ale również aktywnie uczestniczył w ich implementacji, monitorując efekty oraz dostosowując strategię do bieżących potrzeb projektowych. Wdrożone rozwiązania obejmowały m.in. atomizację przypadków testowych, analizę ryzyka, systematyczne wykorzystanie testowania eksploracyjnego, deterministyczną automatyzację oraz wdrożenie hybrydowego *frameworku* testowego opartego na języku Python i zintegrowanego z narzędziami do zarządzania testami oraz środowiskiem ciągłej integracji.

Pozostałe prace, ze względu na ich zakres i złożoność, były realizowane zespołowo przez inżynierów testów, przy czym autor pełnił rolę koordynatora i osoby odpowiedzialnej za nadzór merytoryczny oraz zapewnienie spójności wdrażanych rozwiązań. Współpraca z zespołem obejmowała zarówno planowanie eksperymentów, jak i bieżącą analizę wyników oraz optymalizację procesu testowego w oparciu o zdefiniowane metryki jakościowe.

Specyfika cyklu rozwoju produktu w przedsiębiorstwie oraz charakter wdrożeniowy pracy sprawiły, że nie było możliwości wielokrotnego stosowania różnych technik testowania dla tego samego systemu. Kolejne wersje oprogramowania ulegały ciągłym zmianom, a presja harmonogramu uniemożliwiła ponowne testowanie tych samych funkcjonalności innymi metodami. W konsekwencji porównanie skuteczności strategii testowych mogło odbywać się jedynie w odniesieniu do innych projektów, które — mimo podobieństw — nigdy nie są w pełni identyczne, co nadaje takim analizom charakter przybliżony.

Część wdrożenia realizowana była w środowisku międzynarodowych zespołów rozproszonych. Wdrożenie nowych strategii testowania w zespołach rozproszonych wymaga nie tylko odpowiednich narzędzi i technik, ale przede wszystkim świadomego zarządzania zmianą, inwestycji w kompetencje zespołu oraz systematycznego monitorowania efektów. Kluczowe jest zapewnienie spójności procesów, jasnej komunikacji oraz elastyczności w dostosowywaniu rozwiązań do lokalnych warunków projektowych. Analiza procesu wdrożenia pozwoliła zidentyfikować kluczowe czynniki organizacyjne i techniczne, które sprzyjały skutecznej adaptacji nowych strategii testowania:

1. Jasny podział ról i odpowiedzialności — wyróżnienie roli kierownika projektu testowania funkcjonalnego umożliwiło sprawną komunikację, szybką reakcję na problemy oraz efektywne zarządzanie zmianą.

2. Standaryzacja procesów i narzędzi — zastosowanie jednolitego *frameworku* testowego, zintegrowanego z narzędziem do zarządzania testami oraz współpracującego ze środowiskiem ciągłej integracji, pozwoliło na szybkie wdrożenie nowych praktyk w różnych lokalizacjach i projektach.

3. Atomizacja przypadków testowych i śledzenie powiązań — precyzyjne powiązanie testów z wymaganiami funkcjonalnymi oraz automatyczne raportowanie postępów umożliwiło zespołom rozproszonym łatwe monitorowanie zakresu testów i szybkie reagowanie na zmieniające się wymagania.

4. Szkolenia i rozwój kompetencji — przeprowadzenie dedykowanych szkoleń z obsługi *frameworku*, projektowania testów w oparciu o analizę ryzyka oraz praktyki testowania eksploracyjnego. Nacisk został również położony na przeglądy wzbogacone o dedykowane listy kontrolne. Rozwiązania te znacząco podniosły poziom kompetencji zespołów i ułatwiły adaptację nowych rozwiązań.

5. Synchronizacja prac — plan testów realizowany był w ramach interwałów planowania (PI), co umożliwiło efektywne wykorzystanie wspólnej infrastruktury. Prace konsultowane były z zespołami programistów, aby zminimalizować liczbę blokad w testach.

6. Automatyzacja metryk — zbieranie i wizualizacja metryk w formie pulpitów nawigacyjnych (ang. dashboard) w narzędziu Power BI pozwoliły na bieżące monitorowanie efektywności wdrożenia i szybkie podejmowanie decyzji.

7. Kultura otwartości na zmiany — regularne retrospektywy, przeglądy kodu i projektów testów oraz otwarta komunikacja sprzyjały identyfikacji barier i wdrażaniu usprawnień.

Zakres testów zaczął być określany i kontrolowany przy wykorzystaniu zestawu metryk oraz jasno zdefiniowanych kryteriów wejścia i wyjścia, takich jak współczynnik efektywności testów (TER), gęstość wykrytych błędów (FD), niezawodność automatyzacji (TAR), procent wykrytych defektów (DDP) oraz wykresy spalania (burndown chart). Wskaźniki te umożliwiły bieżące monitorowanie przebiegu kampanii testowej, ocenę skuteczności wdrożonych rozwiązań oraz podejmowanie decyzji dotyczących zakończenia testów. Przykładowo, rozpoczęcie kolejnej iteracji w ramach kampanii testowej było możliwe po spełnieniu warunków takich jak pozytywne przejście testów

promocyjnych w środowisku CI, brak krytycznych defektów bezpieczeństwa oraz dostępność wymaganych obiektów testowych. Zakończenie procesu testowania uzależnione zostało natomiast od osiągnięcia założonego poziomu pokrycia wymagań, spełnienia ustalonych progów metryk oraz braku istotnych defektów.

Ponadto, na bieżąco analizowany był dzienny przyrost nowych przypadków testowych, liczba wykonanych testów (z uwzględnieniem ich rezultatów) oraz liczba wykrytych defektów. Te dane pozwoliły nie tylko śledzić postępy projektu, ale także identyfikować potencjalne przeszkody. Stan projektu oceniany był również na podstawie liczby testów gotowych do uruchomienia w stosunku do tych, które są w fazie projektowania, wymagają aktualizacji, zostały uznane za zbędne lub wyłączone z innych powodów. Automatycznie monitorowany został również poziom pokrycia wymagań funkcjonalnych przez testy.

Jednym z kluczowych usprawnień wdrożonych w ramach strategii testowej podczas realizacji projektu doktorskiego było zastosowanie jednolitej notacji oraz automatycznej generacji przypadków testowych dla parametrów konfiguracyjnych systemów wbudowanych w projektach Oscar i Lima. Specyfikacja wymagań została przygotowana w postaci uporządkowanych plików JSON, które odwzorowywały strukturę urządzeń — podział na klasy, instancje oraz atrybuty umożliwił precyzyjne opisanie parametrów. Dzięki temu podejściu znacząco ograniczono nakład pracy testerów: automatyzacja dokumentacji i implementacji 437 przypadków testowych pozwoliła zaoszczędzić około 146 dni pracy inżynierskiej (przy założeniu dwóch godzin na każdy przypadek i sześciogodzinnym dniu pracy).

Wprowadzono również istotną zmianę w sposobie śledzenia powiązań pomiędzy testami a wymaganiami. W poprzednich projektach jeden przypadek testowy mógł być powiązany z wieloma wymaganiami funkcjonalnymi, co prowadziło do trudności w analizie poprawności implementacji poszczególnych funkcjonalności — zdarzało się, że liczba powiązań przekraczała sto dla pojedynczego testu. W projektach Oscar i Lima wdrożono zasadę tworzenia testów w sposób maksymalnie atomowy, tak aby każdy przypadek testowy obejmował możliwie najmniejszą liczbę wymagań. Efekt tej zmiany był widoczny: dla 99 % testów udało się zachować jednoznaczność powiązań, a w pozostałych przypadkach liczba powiązanych wymagań nie przekroczyła siedmiu.

W projektach Oscar i Lima strategia testowa była ustalana już na etapie opracowywania szczegółowych planów testów dla poszczególnych obszarów, które razem tworzyły główny plan testów. Dokumentacja obejmowała testy funkcjonalne oprogramowania

wbudowanego, testy funkcjonalne oprogramowania, testy sprzętowe, testy systemowe oraz testy związane z cyberbezpieczeństwem. Każdy z tych obszarowych planów zawierał wysokopoziomowe scenariusze, określające zakres testowanych funkcjonalności, bez wchodzenia w szczegóły techniczne ich realizacji.

Nowością wdrożoną w ramach projektu doktorskiego było przypisywanie do każdego scenariusza odpowiedniej akcji w zależności od poziomu ryzyka, jaki został mu przyznany. Dla scenariuszy o niskiej ocenie ryzyka (poniżej 8) rezygnowano z testowania lub ograniczano się do przeglądu kodu. Funkcjonalności o średnim poziomie ryzyka (8–24) były poddawane testom manualnym, testom według listy kontrolnej lub testowaniu eksploracyjnemu. Natomiast dla scenariuszy o wysokim ryzyku (powyżej 24) wdrażano automatyzację testów, co umożliwiało ich regularną weryfikację w ramach testów regresyjnych. W wyjątkowych przypadkach, wynikających z uwarunkowań biznesowych, obowiązujących norm lub dostępnych technologii, rodzaj przypisanej akcji mógł być inny.

Analogiczne podejście zastosowano przy kwalifikowaniu testów regresyjnych. Skala ryzyka pozwalała przypisać testy do jednej z trzech kategorii: codziennej regresji (testy uruchamiane cyklicznie w środowisku CI), regresji jednokrotnej (testy wykonywane raz w cyklu wydania produktu) oraz regresji rozszerzonej, realizowanej w miarę dostępności zasobów lub w odpowiedzi na szczególne potrzeby biznesowe.

Testowanie eksploracyjne (ET) włączono jako usystematyzowane ramy postępowania z kartami sesji (test charters) i ramami czasowymi. Wyniki ET przekształcano w testy skryptowe i — po osiągnięciu wymaganej deterministyki — kwalifikowano do regresji. W projekcie Oscar ET ujawniło defekty trudne do odtworzenia, które najpewniej umknęłyby testom skryptowym, a część z powstałych procedur włączono następnie do regresji i ciągłej integracji w kolejnych wydaniach (podejście przeniesiono również do projektu Lima).

Wdrożenie modułowego *frameworku* testowego, opartego na języku Python i zintegrowanego z narzędziami do zarządzania testami oraz środowiskiem ciągłej integracji, okazało się kluczowym czynnikiem sukcesu w optymalizacji procesu testowania systemów wbudowanych. Hybrydowa architektura łącząca elastyczność podejścia modułowego z możliwością reużywalności kodu oraz oddzieleniem danych testowych od logiki testów, umożliwiła szybkie dostosowanie narzędzi do specyfiki różnych projektów i wymagań. *Framework* zapewnił standaryzację procesu automatyzacji testów, ułatwiając programowanie skryptów, organizację repozytorium oraz raportowanie wyników. Dzięki

zastosowaniu wspólnych bibliotek komunikacyjnych i testowych, a także kontekstowych modułów dedykowanych dla konkretnych urządzeń, możliwe było efektywne zarządzanie przypadkami testowymi oraz ich szybka adaptacja do zmian w projekcie. Integracja z narzędziem do zarządzania testami przez dedykowane API pozwoliła na automatyczne śledzenie powiązań z wymaganiami oraz generowanie raportów. Wprowadzenie standardów kodowania, a także automatycznej generacji dokumentacji technicznej (za pomocą narzędzia Sphinx), podniosło czytelność i utrzymywalność kodu testowego. Wdrożony *framework* testowy stał się fundamentem nowoczesnego procesu weryfikacji systemów wbudowanych w przedsiębiorstwie, umożliwiając nie tylko efektywną automatyzację testów, ale także zwinne zarządzanie zakresem i jakością testowania w dynamicznie zmieniających się warunkach projektowych.

W zakresie zgodności procesowej cała opisywana metoda pozostaje spójna z wymaganiami dokumentacyjnymi normy ISO/IEC/IEEE 29119 (warstwowanie planowania i raportowania, macierz ryzyka) oraz z praktykami RAPL w obszarze Verification & Validation (śledzenie powiązań, kryteria wejścia/wyjścia, testy promocyjne jako bramki jakościowe). Taka konstrukcja ułatwiła standaryzację raportów i porównywalność wyników między projektami, także prowadzonymi w ramach innych jednostek biznesowych.

W badaniach prowadzonych w warunkach przemysłowych, bez ścisłej kontroli eksperymentalnej, niezwykle trudno jest wyizolować wpływ pojedynczej zmiany na przebieg procesu. Wiele czynników zmienia się równocześnie, co stwarza ryzyko błędnej atrybucji przyczynowo-skutkowej. Na obserwowany wzrost gęstości defektów FD oraz współczynnika efektywności testowania TER mogły mieć, przynajmniej częściowo, wpływ czynniki zakłócające:

1. Nowość i złożoność produktu — nowe, niedojrzałe produkty i architektury (jak w przypadku projektów Oscar i Lima) są z natury bardziej podatne na defekty, niż systemy rozwijane od dłuższego czasu. Wysoka liczba wykrytych błędów może więc częściowo odzwierciedlać niższą początkową jakość kodu i złożoność produktów.

2. Jednoczesna zmiana wielu elementów strategii testowej oraz narzędzi — wpływ wprowadzenia zmian w strategii testowej oraz nowego *frameworku* testowego nie został rozdzielony. Bez takiego rozróżnienia nie można określić, jaka część zaobserwowanej poprawy wynika z lepszego narzędzia, a jaka z wdrożenia nowej, wysoce skutecznej strategii testowej.

3. Skład i doświadczenie zespołu — praca nie obejmuje zagadnień związanych z kapitałem ludzkim, takich jak umiejętności, doświadczenie, czy motywacja zespołów inżynierskich przypisanych do projektów.

Mimo powyższych ograniczeń, wdrożenie przyniosło namacalne efekty techniczne i organizacyjne: skrócenie czasu kampanii dzięki automatyzacji i atomizacji, zwiększenie wykrywalności (TER) i gęstości błędów (FD) dla projektów wzorcowych (Oscar, Lima) na tle grup referencyjnych, a także większą stabilność procesu dzięki rygorystycznym progom kwalifikacji do regresji i ciągłemu monitorowaniu metryk. W efekcie, proponowane przez autora techniki testowania stały się fundamentem wdrożonej strategii testowej, a ich implementacja przyczyniła się do znaczącego wzrostu efektywności testowania oraz poprawy jakości dostarczanych rozwiązań. Opracowane ramy postępowania testowego zostały uznane za wytyczne procesowe i wdrożone jako standard w przedsiębiorstwie, potwierdzając zasadność przyjętych rozwiązań oraz wskazując kierunek dalszego rozwoju praktyk testowania w sektorze systemów wbudowanych.

6.3 Potencjalny wpływ wyników na przyszłość testowania

Przeprowadzone badania oraz wdrożenia w projektach Oscar i Lima dowodzą, że integracja podejścia opartego na analizie ryzyka, atomizacji przypadków testowych, testowania eksploracyjnego oraz deterministycznej automatyzacji pozwala na znaczącą optymalizację procesu testowego. Takie ramy postępowania nie tylko zwiększają efektywność wykrywania defektów, ale również podnoszą jakość końcowego produktu oraz umożliwiają bardziej racjonalne gospodarowanie zasobami zespołu testerskiego.

Jednym z kluczowych wniosków płynących z pracy jest potrzeba dalszego doskonalenia narzędzi i technik automatyzacji testów. W szczególności, wdrożony hybrydowy *framework* testowy oparty o język Python oraz integrację z narzędziami do zarządzania testami i środowiskiem ciągłej integracji stanowi solidną bazę do dalszych eksperymentów. Modularność środowiska jest jednym z najważniejszych czynników wpływających na efektywność i skalowalność procesu testowania. Dalsze podzielenie *frameworku* na mniejsze, reużywalne moduły objęte systemem wersjonowania, wymagałoby pracy związanej z definicją interfejsów pomiędzy nimi, ale jednocześnie pozwoliłoby na niezależne rozwijanie, testowanie i utrzymywanie poszczególnych komponentów. W prak-

tyce oznacza to, że biblioteki komunikacyjne, testowe oraz kontekstowe mogłyby być wykorzystywane w różnych projektach, niezależnie od zmian w architekturze urządzenia czy protokołach komunikacyjnych. Takie podejście umożliwia szybkie dostosowanie narzędzi do nowych wymagań, minimalizuje ryzyko błędów propagowanych pomiędzy testami oraz znacząco skraca czas wdrożenia nowych funkcjonalności. Modularność sprzyja także efektywności zespołów testerskich — raz napisane komponenty mogą być wielokrotnie używane w różnych projektach, co generuje wymierne oszczędności czasowe i finansowe. W systemach wbudowanych, gdzie testy muszą uwzględniać zarówno warstwę sprzętową, jak i programową, modularny *framework* pozwala na szybkie przełączanie się pomiędzy testami różnych typów (np. funkcjonalnych, wydajnościowych, bezpieczeństwa) oraz łatwą integrację z narzędziami do zarządzania testami i środowiskami ciągłej integracji.

Kolejnym krokiem powinno być także dobranie strategii migracji z poprzedniego *frameworku* do nowego narzędzia. Plan przejścia powinien obejmować priorytetyzację testów i ich kwalifikację do jednej z trzech kategorii:

1. Testy o wysokiej wartości regresyjnej.
2. Testy obszarów funkcjonalnych o niskiej zmienności.
3. Testy o niskiej wartości informacyjnej lub chronicznie niestabilne.

Kryteria decyzyjne powinny objąć koszta utrzymania, czas wykonania, podatność na automatyzację, wartość w regresji oraz ryzyka powiązane z danym obszarem funkcjonalnym. Testy kategorii pierwszej powinny zostać w pierwszej kategorii przepisane w języku Python. Testy kategorii drugiej mogą wykorzystać mostkowanie, czyli wywołanie istniejących sekwencji z poziomu Pythona, a testy kategorii trzeciej powinny zostać stopniowo wygaszane i wycofywane.

Sztuczna inteligencja (SI) rewolucjonizuje automatyzację testów, szczególnie w obszarze systemów wbudowanych, gdzie tradycyjne metody często okazują się niewystarczające. Przyszłe badania mogą koncentrować się na rozwoju mechanizmów automatycznego generowania przypadków testowych z wykorzystaniem dużych modeli językowych oraz technik generowania wspomaganego wyszukiwaniem, które pozwalają dynamicznie wstrzykiwać aktualne informacje do procesu generowania testów. Wykorzystanie uczenia przez wzmocnienie w ocenie jakości generowanych testów może dodatkowo zwiększyć powtarzalność i trafność wyników. Możliwa jest także analiza wyników

testów i identyfikacja powtarzalnych problemów, co przyspieszy diagnozowanie i naprawę błędów, a także predykcyjne wykrywanie defektów, które mogą być trudne do zauważenia przez inżynierów testów. Przykłady wdrożeń pokazują, że SI pozwala na skrócenie cyklu wytwarzania i utrzymania skryptów, zwiększenie elastyczności projektu oraz lepsze dostosowanie do zmieniających się wymagań klientów.

Osobną kwestią jest testowanie generatywnej SI będącej częścią funkcjonalności systemu wbudowanego, na przykład w formie dużego modelu językowego będącego asystentem użytkownika albo elementem wykorzystywanym do predykcji awarii. Testowanie sztucznej inteligencji jest bardzo kosztowne obliczeniowo i energetycznie, a naprawa błędów jest trudna i tworzy nowe wyzwania w określeniu zakresu testów regresyjnych. Funkcjonalność modeli ze względu na zjawisko halucynacji nie jest niezawodna, a ten sam proces generowania treści uruchomiony wielokrotnie charakteryzuje rozrzut wyników.

W pracy podkreślono wagę systematycznego podejścia do oceny efektywności metod testowych, w tym stosowania standaryzowanych metryk takich jak TER, FD, TAR czy DDP. Dalszy rozwój tej dziedziny powinien obejmować prace nad ujednoliceniem metryk oraz ich wdrożeniem w różnych organizacjach, co umożliwi obiektywne porównywanie skuteczności różnych strategii testowych. Standaryzacja raportowania i automatyzacja zbierania danych, np. z wykorzystaniem narzędzi typu Power BI, pozwoli na bardziej świadome podejmowanie decyzji oraz szybszą adaptację dobrych praktyk w branży. Metryki te można w dalszym stopniu rozbudować o dane dotyczące defektów pochodzących ze środowiska produkcyjnego, czyli bezpośrednio od klientów i użytkowników końcowych, co pozwoli jeszcze lepiej odnieść się do efektywności procesów testowych w organizacji.

Możliwe jest również analizowanie przyczyn występowania błędów (ang. Root Cause Analysis, RCA) poprzez systematyczną identyfikację pierwotnych przyczyn defektów lub usterek w systemie. W testowaniu systemów wbudowanych RCA pozwala nie tylko na wykrycie błędu, ale przede wszystkim na zrozumienie, dlaczego on wystąpił — czy wynika to z wadliwej implementacji, problemów sprzętowych, błędów w komunikacji, czy może nieprecyzyjnej specyfikacji wymagań. Proces polega na analizie raportów testowych i identyfikowaniu powtarzalnych wzorców awarii (na przykład błędów pojawiających się w określonych warunkach środowiskowych lub przy specyficznych konfiguracjach sprzętu). RCA pozwala na wypracowanie trwałych rozwiązań, dzięki którym eliminowane są źródła problemów, co przekłada się na wzrost jakości produktu i re-

dukcję kosztów związanych z późniejszymi poprawkami, co widoczne było na Rysunku 2.4.

Testowanie oparte na analizie ryzyka wdrożone jako technika priorytetyzacji testów może być w dalszych etapach rozbudowane o analizę rodzajów i skutków błędów (ang. Failure Mode and Effects Analysis, FMEA). Jest to technika, która pozwala na ocenę potencjalnych awarii i ich skutków dla działania systemu. Na etapie projektowania testów identyfikowane są wszystkie możliwe sposoby, w jakie dany komponent systemu może ulec awarii. Każdy tryb awarii jest oceniany pod kątem prawdopodobieństwa wystąpienia, wykrywalności oraz potencjalnych skutków dla użytkownika lub systemu — takich jak utrata funkcjonalności, czy zagrożenie bezpieczeństwa. Wyniki FMEA służą jako dane wejściowe do priorytetyzacji testów.

Także testowanie eksploracyjne, wdrożone jako usystematyzowana technika, okazało się szczególnie skuteczne w wykrywaniu defektów trudnych do odtworzenia oraz luk w pokryciu testowym. Przyszłe badania mogą skupić się na dalszym rozwijaniu tej metody, zwłaszcza w kontekście projektów o wysokiej zmienności wymagań oraz braku pełnej specyfikacji funkcjonalnej. Możliwe jest także rozszerzenie zastosowania testów eksploracyjnych na inne typy systemów wbudowanych oraz integracja ich wyników z automatycznymi testami regresyjnymi, co pozwoli na jeszcze większą elastyczność i efektywność procesu testowego.

Wykorzystana metodologia studium przypadku obejmuje procesy i projekty realizowane w jednym przedsiębiorstwie — Rockwell Automation. Eksperymenty wdrożeniowe, stanowiące fundament analizy empirycznej, skoncentrowano na celowo wybranej, nielosowej próbie projektów (Oscar, Lima, Quebec i Sierra). Taki dobór metody badawczej pozwolił na głęboką analizę złożonych zjawisk w ich naturalnym otoczeniu. Jednakże ta sama cecha jest jednocześnie źródłem najpoważniejszego ograniczenia — braku walidacji zewnętrznej, czyli ograniczonej możliwości generalizacji wyników na inne organizacje, projekty, czy domeny technologiczne.

Sukces wdrożonego *frameworku* jest nierozerwalnie związany z warunkami panującymi w firmie. Należą do nich unikalna kultura organizacyjna, procesy rozwoju produktu, wykorzystanie *Scaled Agile Framework* do organizacji pracy, czy obecność długu technicznego w postaci wewnętrznie rozwijanego, starszego środowiska testowego. Sukces rozwiązania opartego o język Python, mierzony wzrostem metryk TER i FD, może więc wynikać z faktu, że stanowi on znaczące usprawnienie w porównaniu do stanu zastanego. Nie znaczy to, że *framework* ten jest obiektywnie lepszy od innych

nowoczesnych, komercyjnych lub otwartych narzędzi testowych. Przeprowadzone prace świadczą zatem o skuteczności lokalnej optymalizacji procesu w ściśle określonych warunkach.

Należy mieć na uwadze, iż wpływ na uzyskane wyniki może mieć także zmienność zespołów testerskich i programistycznych między projektami, a także różnice w dojrzałości kodu, narzędzi i dostępności sprzętu, które mogą bezpośrednio rzutować na wyniki testów. Zatem rezultaty uzyskane w środowisku Rockwell Automation mogą nie być w pełni odtwarzalne w innych organizacjach o odmiennych procesach i narzędziach. Przyszłe badania mogą rozszerzyć próbę poddaną analizie statystycznej oraz obejmować inne środowiska przemysłowe.

Proponowany *framework* testowy można rozpatrywać także w kategorii praktycznego rozwiązania problemów lub konkretnej implementacji praktyk, które związane są z wdrażaniem modeli dojrzałości procesów testowych takich jak *Test Maturity Model integration* (TMMi) czy *Test Process Improvement Next* (TPI Next). TMMi to ustrukturyzowany, pięciopoziomowy model, który pozwala ocenić procesy testowe w organizacji. Na drugim poziomie dojrzałości wymaga zdefiniowania polityki i strategii testowej oraz planowania testów, co w pracy doktorskiej realizowane jest poprzez testowanie oparte na wymaganiach i analizie ryzyka. Na poziomie trzecim TMMi oczekuje integracji testowania z cyklem życia oprogramowania i standaryzacji procesów, na co odpowiedzią jest *framework* testowy zintegrowany ze środowiskiem ciągłej integracji i narzędziem do zarządzania testowaniem. Wymagany na poziomie czwartym pomiar jakości i efektywności realizowany jest poprzez zdefiniowanie i systematyczne stosowanie metryk. Wdrożone rozwiązania bezpośrednio adresują także kilka kluczowych obszarów modelu TPI Next, takich jak strategia testowa, zarządzanie defektami, środowisko testowe oraz metryki. Dalsze prace mogą w dalszym stopniu rozszerzać i eksplorować ten wątek.

Finalnie, wyniki pracy mogą mieć również istotny wpływ na przyszłość certyfikacji testerskiej, w tym na aktualizację sylabusów ISTQB oraz rozwój nowych specjalizacji dedykowanych testerom systemów wbudowanych. Wprowadzenie nowych metod i narzędzi do programów certyfikacyjnych pozwoli testerom na zdobycie bardziej wyspecjalizowanej wiedzy i umiejętności, co przyczyni się do poprawy jakości i niezawodności testowanych systemów. Dalsze badania mogą również wspierać rozwój materiałów edukacyjnych oraz programów szkoleniowych, które będą odpowiadać na aktualne wyzwania.

Bibliografia

[1] Embedded systems market size, share & covid-19 impact analysis, by component (hardware and software), by type (standalone embedded systems, real-time embedded systems, network embedded systems, and mobile embedded systems), by application (consumer electronics, medical equipment, industrial automation, automotive systems, aerospace & defense technologies, telecommunications, smart devices, and others), and regional forecast, 2023-2030. `https://web.archive.org/web/20240725171213/https://www.fortunebusinessinsights.com/embedded-systems-market-108767`. Dostęp: 2024-07-25.

[2] Apostolos N. Meliones. A comprehensive development guide for network embedded systems. In *2010 8th Workshop on Intelligent Solutions in Embedded Systems*, pages 43–48, 2010.

[3] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems 2nd Edition*. O'Reilly, Sebastopol, 2008.

[4] Rahul Gore, Hariram Satheesh, and Mallikarjun Kande. Platform analysis in embedded systems. In *2014 2nd International Conference on Emerging Technology Trends in Electronics, Communication and Networking*, pages 1–6, 2014.

[5] J.M. Moya, F. Moya, and J.C. Lopez. A flexible approach to the design of complex embedded systems. In *Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat. No.01TH8558)*, pages 237–241, 2001.

[6] A.A. Jerraya. Long term trends for embedded system design. In *Euromicro Symposium on Digital System Design, 2004. DSD 2004.*, pages 20–26, 2004.

[7] Techinsights: Samsung's 3nm gaa process identified in a crypto-mining asic designed by china startup microbt. `https://web.archive.org/web/`

20240927202238/https://www.digitimes.com/news/a20230718VL203/samsung-china-3nm-asic.html. Dostęp: 2024-09-27.

[8] International roadmap for devices and systems™ 2021 update: More moore. https://irds.ieee.org/editions/2021/more-moore. Dostęp: 2024-12-15.

[9] Janusz Rajski, Vivek Chickermane, Jean-François Côté, Stephan Eggersglüß, Nilanjan Mukherjee, and Jerzy Tyszer. The future of design for test and silicon lifecycle management. *IEEE Design & Test*, 41(4):35–49, 2024.

[10] M. Youssef, Sungjoo Yoo, A. Sasongko, Y. Paviot, and A.A. Jerraya. Debugging hw/sw interface for mpsoc: video encoder system design case study. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 908–913, 2004.

[11] Cheng Pang, Wenbin Dai, Qingdi Miao, Jinxian Liang, Guoqing Cai, Shu Lu, and Valeriy Vyatkin. Software-defined automation and control a preliminary study. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 5497–5502, 2017.

[12] Darja Smite, Nils Moe, and Pär Ågerfalk. *Agility Across Time and Space*. Springer Verlag, Berlin, 01 2010.

[13] 16 amazing agile statistics [2023]: What companies use agile methodology. https://web.archive.org/web/20241215210247/https://www.zippia.com/advice/agile-statistics/. Dostęp: 2024-12-15.

[14] J. Nawrocki, B. Walter, and A. Wojciechowski. Toward maturity model for extreme programming. In *Proceedings 27th EUROMICRO Conference. 2001: A Net Odyssey*, pages 233–239, 2001.

[15] Słownik terminów testowych istqb. https://glossary.istqb.org/pl_PL/term/testowanie/2. Dostęp: 2024-12-29.

[16] Lech Madeyski and Marcin Kawalerowicz. Continuous defect prediction: The idea and a related dataset. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 515–518, 2017.

[17] Szymon Stradowski and Lech Madeyski. Can we knapsack software defect prediction? nokia 5g case. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 365–369, 2023.

[18] Szymon Stradowski and Lech Madeyski. Bridging the gap between academia and industry in machine learning software defect prediction: Thirteen considerations. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1098–1110, 2023.

[19] Szymon Stradowski and Lech Madeyski. Interpretability/explainability applied to machine learning software defect prediction: An industrial perspective. *IEEE Software*, 42(3):125–132, 2025.

[20] Ghadeer Murad, Aalaa Badarneh, Abdallah Qusef, and Fadi Almasalha. Software testing techniques in iot. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 17–21, 2018.

[21] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley Publishing, 2nd edition, 2018.

[22] Michał Kaczmarek and Przemysław Koralewicz. Hardware in the loop simulations of industrial application using system on the chip architecture. In *2016 International Conference on Signals and Electronic Systems (ICSES)*, pages 157–160, 2016.

[23] ISO/IEC/IEEE. Ieee/iso/iec international standard for software and systems engineering–software testing–part 3:test documentation - redline. *ISO/IEC/IEEE 29119-3:2021(E) - Redline*, pages 1–274, 2021.

[24] Stefan Arthofer, Stefan Wilker, and Thilo Sauter. Development of a test automation framework based on a comparison of different approaches for test automation in the embedded systems area. In *2024 IEEE 7th International Conference on Industrial Cyber-Physical Systems (ICPS)*, pages 1–6, 2024.

[25] Tabinda Sarwar, Wajiha Habib, and Fahim Arif. Requirements based testing of software. In *2013 Second International Conference on Informatics & Applications (ICIA)*, pages 347–352, 2013.

[26] Andrei Contan, Catalin Dehelean, and Liviu Miclea. Test automation pyramid from theory to practice. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5, 2018.

[27] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley, London, 2002.

[28] Paul C Jorgensen. *Modeling software behavior: a craftsman's approach*. CRC press, 2009.

[29] Pramod Mathew Jacob and M. Prasanna. A comparative analysis on black box testing strategies. In *2016 International Conference on Information Science (ICIS)*, pages 1–6, 2016.

[30] Matthias Hamburg and Adam Roman. Black-box testing for practitioners: A case of the new istqb test analyst syllabus. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 634–645, 2025.

[31] L.J. White and E.I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, 1980.

[32] L.A. Clarke, J. Hassell, and D.J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, SE-8(4):380–390, 1982.

[33] Mamta Sharma and Subhash Chandra B. Automatic generation of test suites from decision table - theory and implementation. In *2010 Fifth International Conference on Software Engineering Advances*, pages 459–464, 2010.

[34] K.-T. Cheng and J.-Y. Jou. Functional test generation for finite state machines. In *Proceedings. International Test Conference 1990*, pages 162–168, 1990.

[35] Vaclav Rechtberger, Miroslav Bures, and Bestoun S. Ahmed. Overview of test coverage criteria for test case generation from finite state machines modelled as directed graphs. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 207–214, 2022.

[36] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. Metamorphic testing: Testing the untestable. *IEEE Software*, 37(3):46–53, 2020.

[37] Tsong Yueh Chen. Metamorphic testing: A simple method for alleviating the test oracle problem. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 53–54, 2015.

[38] Zenghui Zhou, Zheng Zheng, Tsong Yueh Chen, Jinyi Zhou, and Kun Qiu. Follow-up test cases are better than source test cases in metamorphic testing: A preliminary study. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*, pages 69–74, 2021.

[39] Altaf Hussain, Aamer Nadeem, and Muhammad Touseef Ikram. Review on formalizing use cases and scenarios: Scenario based testing. In *2015 International Conference on Emerging Technologies (ICET)*, pages 1–6, 2015.

[40] Javier J. Gutierrez, Maria J. Escalona, Manuel Mejias, Jesus Torres, and Arturo H. Centeno. A case study for generating test cases from use cases. In *2008 Second International Conference on Research Challenges in Information Science*, pages 209–214, 2008.

[41] Bill Hasling, Helmut Goetz, and Klaus Beetz. Model based testing of system requirements using uml use case models. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 367–376, 2008.

[42] Lahbib Naimi, El Mahi Bouziane, Mohamed Manaouch, and Abdeslam Jakimi. A new approach for automatic test case generation from use case diagram using llms and prompt engineering. In *2024 International Conference on Circuit, Systems and Communication (ICCSC)*, pages 1–5, 2024.

[43] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Softw. Eng.*, 50(4):911–936, April 2024.

[44] Marcin Bajer, Marek Szlagor, and Marek Wrzesniak. Embedded software testing in research environment. a practical guide for non-experts. In *2015 4th Mediterranean Conference on Embedded Computing (MECO)*, pages 100–105, 2015.

[45] Dietmar Winkler, Reinhard Hametner, Thomas Östreicher, and Stefan Biffl. A framework for automated testing of automation systems. In *2010 IEEE 15th Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, pages 1–4, 2010.

[46] Florian Muttenthaler, Stefan Wilker, and Thilo Sauter. Lean automated hardware/software integration test strategy for embedded systems. In *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, volume 1, pages 783–788, 2021.

[47] Tarun Chatterjee. Test lead time and cost improvement of automotive embedded project - a new perspective with automation and ci/cd. In *2024 International Conference on Vehicular Technology and Transportation Systems (ICVTTS)*, volume 1, pages 1–6, 2024.

[48] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., USA, 1990.

[49] Sayed Abdelgaber, Rasha Mansour Mohamed, Laila Abdel Hamid, and A. Abdo. Ontological approach for overcoming pesticide paradox in inter-class testing of object-oriented applications. *Journal of Theoretical and Applied Information Technology*, 100(9):2772–2790, May 2022.

[50] Pawel Kubczak, Wiktor Wozniak, Jakub Nikonowicz, Lukasz Matuszewski, and Mieczyslaw Jessa. An online platform for testing and evaluating random number generators. In *2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, 2021.

[51] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[52] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[53] Harshit Kumar Chaubey, Gaurav Tripathi, Rajnish Ranjan, and Srinivasa k. Gopalaiyengar. Comparative analysis of rag, fine-tuning, and prompt engineering in chatbot development. In *2024 International Conference on Future Technologies for Smart Society (ICFTSS)*, pages 169–172, 2024.

[54] MR Bhavya, Anish Damodaran, and Sindhu Ranganath. An ai based smart test case generator for embedded device. In *2022 Second International Conference on Power, Control and Computing Technologies (ICPC2T)*, pages 1–5, 2022.

[55] Ravi Prakash Verma and Md. Rizwan Beg. Generation of test cases from software requirements using natural language processing. In *2013 6th International Conference on Emerging Trends in Engineering and Technology*, pages 140–147, 2013.

[56] Mert Yurdakul. Bugle: Method for duplicate defect report detection with transformers. `https://web.archive.org/web/20250320135529/https://testscouts.se/bugle-method-for-duplicate-defect-report-detection-with-transformers`. Dostęp: 2025-03-20.

[57] Struktura certyfikacji istqb® 2025 po polsku. `https://testerzy.pl/news/flash/struktura-certyfikacji-istqb-2025-po-polsku`. Dostęp: 2025-10-03.

[58] Attila Szatmári, Tamás Gergely, and Árpád Beszédes. Istqb-based software testing education: Advantages and challenges. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 389–396, 2023.

[59] Istqb - our certifications. `https://web.archive.org/web/20241117051246/https://www.istqb.org/certifications/certification-list/`. Dostęp: 2024-11-17.

[60] Scaled Agile Framework. Framework – scaled agile framework. `https://web.archive.org/web/20250516094528/https://framework.scaledagile.com`. Dostęp: 2025-05-16.

[61] R.K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009.

[62] Julian M. Bass, Sarah Beecham, and John Noll. Experience of industry case studies: A comparison of multi-case and embedded case study methods. In *2018 IEEE/ACM 6th International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 13–20, 2018.

[63] J.M. Verner, J. Sampson, V. Tosic, N.A. Abu Bakar, and B.A. Kitchenham. Guidelines for industrially-based multiple case studies in software engineering. In *2009 Third International Conference on Research Challenges in Information Science*, pages 313–324, 2009.

[64] Liang Yu. Ethical considerations in case studies. In *2020 1st Workshop on Ethics in Requirements Engineering Research and Practice (REthics)*, pages 15–21, 2020.

[65] Ellen Souza, Cristine Gusmao, Keldjan Alves, Julio Venancio, and Renata Melo. Measurement and control for risk-based test cases and activities. In *2009 10th Latin American Test Workshop*, pages 1–6, 2009.

[66] Ellen Souza, Cristine Gusmão, and Júlio Venâncio. Risk-based testing: A case study. In *2010 Seventh International Conference on Information Technology: New Generations*, pages 1032–1037, 2010.

[67] Juergen Grossmann, Michael Felderer, Johannes Viehmann, and Ina Schieferdecker. A taxonomy to assess and tailor risk-based testing in recent testing standards. *IEEE Software*, 37(1):40–49, 2020.

[68] Armin Beer and Rudolf Ramler. The role of experience in software testing practice. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 258–265, 2008.

[69] Jiujiu Yu, Jishan Zhang, Liqiong Pan, Yun Chen, Ning Wu, and Wenling Sun. Software exploratory testing: Present, problem and prospect. In *2021 3rd International Academic Exchange Conference on Science and Technology Innovation (IAECST)*, pages 44–47, 2021.

[70] V Prakash and S Gopalakrishnan. Testing efficiency exploited: Scripted versus exploratory testing. In *2011 3rd International Conference on Electronics Computer Technology*, volume 3, pages 168–172, 2011.

[71] Rafal Kimla and Robert Czerwinski. A methodical approach to functional exploratory testing for embedded systems. *Applied Sciences*, 12(19), 2022.

[72] R. Black. *Advanced Software Testing Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager, 2nd Edition.* Rocky Nook, Santa Barbara, 2014.

[73] S. Kan. *Metrics and Models in Software Quality Engineering, Second Edition.* Addison Wesley, Boston, 2002.

[74] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2014.

[75] K. Someoliayi, S. Jalali, M. Mahdieh, and S. Mirian-Hosseinabadi. Program state coverage: A test coverage metric based on executed program states. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 584–588, Hangzhou, 2019. IEEE.

[76] Rafał Kimla. Overview of metrics applicable in embedded systems functional testing. In *International Conference of Computational Methods in Sciences and Engineering ICCMSE 2021*, volume 2611, page 070004, 11 2022.

[77] ISO/IEC. Systems and software engineering – life cycle processes – requirements engineering, 2018.

[78] Fayona Cowperthwaite, Jennifer Horkoff, and Sylwia Kopczyńska. The effects of native language on requirements quality. In *2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 913–917, 2023.

[79] Michael Felderer and Rudolf Ramler. Integrating risk-based testing in industrial test processes. *Software Quality Journal*, 22, 09 2013.

[80] Michael Felderer and Rudolf Ramler. Experiences and challenges of introducing risk-based testing in an industrial project. In *International Conference on Software Quality*, volume 133, pages 10–29, 01 2013.

[81] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., USA, 2nd edition, 1999.

[82] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., USA, 2001.

[83] W. Makondo, R. Nallanthighal, I. Mapanga, and P. Kadebu. Exploratory test oracle using multi-layer perceptron neural network. In *Intl. Conference on Advances in Computing, Communications and Informatics (ICACCI)*, volume Sept. 21-24, Jaipur, India, 2016.

[84] V. De Oliveira Neves, M.E. Delamaro, and P.C. Masiero. An environment to support structural testing of autonomous vehicles. In *Brazilian Symposium on Computing Systems Engineering*, 2014.

[85] Juha Itkonen, Mika V. Mäntylä, and Casper Lassenius. The role of the tester's knowledge in exploratory software testing. *IEEE Transactions on Software Engineering*, 39(5):707–724, 2013.

[86] Torvald Mårtensson, Daniel Ståhl, Antonio Martini, and Jan Bosch. Efficient and effective exploratory testing of large-scale software systems. *Journal of Systems and Software*, 174:110890, 2021.

[87] Y. Jiujiu, P. Liqiong, W. Ning, Z. Jishan, C. Yun, and S. Wenling. Software exploratory testing: Present, problem and prospect. In *3rd International Academic Exchange Conference on Science and Technology Innovation (IAECST)*, 2021.

[88] Ahmad Nauman Ghazi, Kai Petersen, Elizabeth Bjarnason, and Per Runeson. Levels of exploration in exploratory testing: From freestyle to fully scripted. *IEEE Access*, 6:26416–26423, 2018.

[89] James Bach. Exploratory testing. https://www.satisfice.com/exploratory-testing. Dostęp: 2022-08-17.

Dodatek A

Wykaz symboli i oznaczeń

Symbole, metryki i oznaczenia matematyczne

Symbol	Opis
R	Ocena ryzyka w planowaniu/testowaniu; iloczyn prawdopodobieństwa i konsekwencji: $R = P \times K$.
P	Prawdopodobieństwo wystąpienia ryzyka (skala 1–10 w analizie ryzyka).
K	Konsekwencje (skutki) materializacji ryzyka (skala 1–10).
TER	*Test Effectiveness Ratio* — współczynnik efektywności testu/projektu. W wariancie per-test: $TER = \frac{D_T}{D_A} \times 100\%$; w wariancie porównawczym per-projekt: $TER = \frac{D_P}{D_R} \times 100\%$.
FD	*Fault Density* — gęstość defektów: $FD = \frac{D_A}{L_T} \times 100\%$.
DDP	*Defect Detection Percentage* — procent wykrytych defektów: $DDP = \frac{D_A}{D_A + D_N} \times 100\%$.
TAR	*Test Automation Reliability* — niezawodność automatyzacji testów: $TAR = \left(1 - \frac{R_F}{L_A}\right) \times 100\%$.
$\mathbf{D_T}$	Liczba defektów wykrytych przez dany test.
$\mathbf{D_A}$	Łączna liczba wykrytych defektów (w rozpatrywanym zbiorze).

ciąg dalszy na następnej stronie

Symbol	Opis
$\mathbf{D_P}$	Liczba defektów wykrytych w projekcie (dla wariantu $TER = \frac{D_P}{D_R}$).
$\mathbf{D_R}$	Łączna liczba defektów w projektach referencyjnych (tej samej kategorii).
$\mathbf{D_N}$	Liczba defektów niewykrytych podczas testów (wykrytych dopiero w eksploatacji).
$\mathbf{L_T}$	Liczba wszystkich wykonanych testów.
$\mathbf{R_F}$	Liczba wyników fałszywie pozytywnych i fałszywie negatywnych w automatyzacji.
$\mathbf{L_A}$	Liczba testów automatycznych.
$\boldsymbol{\mu_k^l}$	*TEI* — indeks efektywności testu (wariant zdefiniowany funkcyjnie dla rundy l, modułu k); wartości zależne od zmian $D_k^l(T_i)$ pomiędzy rundami.
$\mathbf{D_k^l(T_i)}$	Liczba defektów typu T_i w module k wykrytych w rundzie l.
$\mathbf{T_1}$	Typ defektu: funkcjonalny.
$\mathbf{T_2}$	Typ defektu: bezpieczeństwa.
$\mathbf{T_3}$	Typ defektu: wydajnościowy.
$\mathbf{T_4}$	Typ defektu: interfejsu (UI/komunikacja).
$\mathbf{r}$	Współczynnik korelacji Pearsona (analiza statystyczna zależności).
$\boldsymbol{\rho}$	Współczynnik korelacji rang Spearmana.
$\mathbf{p}$	Wartość istotności statystycznej (p-value).

Skróty i akronimy

Skrót	Rozwinięcie / opis
API	*Application Programming Interface* — interfejs programistyczny.
ASIC	*Application-Specific Integrated Circuit*.
CI	*Continuous Integration* — ciągła integracja.
CD	*Continuous Deployment/Delivery* — ciągłe wdrażanie/dostarczanie.

ciąg dalszy na następnej stronie

Skrót	Rozwinięcie / opis
CTFL	*Certified Tester Foundation Level* — poziom podstawowy ISTQB.
DDP	*Defect Detection Percentage* (por. sekcja symboli).
DevOps	*Development & Operations* — filozofia łączenia rozwoju i operacji.
ET	*Exploratory Testing* — testowanie eksploracyjne.
FMEA	*Failure Mode and Effects Analysis*.
FD	*Fault Density* (por. sekcja symboli).
GPU	*Graphics Processing Unit*.
GUI	*Graphical User Interface*.
HiL	*Hardware-in-the-Loop*.
IoT	*Internet of Things*.
ISTQB	*International Software Testing Qualifications Board*.
JSON	Format danych: *JavaScript Object Notation*.
CSV	Format danych: *Comma-Separated Values*.
JSONL	Format danych: *JSON Lines*.
LD	*Ladder Diagram* — język drabinkowy sterowników PLC.
LLM	*Large Language Model*.
MBT	*Model-Based Testing*.
MMU	*Memory Management Unit*.
MR	*Metamorphic Relations* — relacje metamorficzne (testowanie metamorficzne).
MPSoC	*Multiprocessor System-on-a-Chip*.
NoC	*Network-on-a-Chip*.
NLP	*Natural Language Processing*.
PI	*Planning Interval* — interwał planowania w SAFe.
PLC	*Programmable Logic Controller*.
RAG	*Retrieval-Augmented Generation*.
RAPL	*Rockwell Automation Product Lifecycle* (proces wytwarzania produktu obejmujący m.in. obszar Verification & Validation).
RTOS	*Real-Time Operating System*.
SAFe	*Scaled Agile Framework*.
SDA	*Software-Defined Automation*.
SiL	*Software-in-the-Loop*.

ciąg dalszy na następnej stronie

Skrót	Rozwinięcie / opis
SoC	*System-on-a-Chip*.
TAF	*Test Automation Framework* — wewnętrzne środowisko automatyzacji (historyczne).
TAR	*Test Automation Reliability* (por. sekcja symboli).
TEI	*Test Effectiveness Index* (por. μ_k^l).
TER	*Test Effectiveness Ratio* (por. sekcja symboli).
TMMi	*Test Maturity Model integration*.
TPI Next	*Test Process Improvement Next*.
UML/SysML	*Unified Modeling Language / Systems Modeling Language*.

Noty redakcyjne

- Definicje metryk TER, FD, DDP, TAR i indeksu μ_l^k odpowiadają formułom użytym w treści rozprawy; symbole składowe ($D_T, D_A, D_P, D_R, L_T, D_N, R_F, L_A$) zebrano tutaj dla wygody czytelnika.
- Wykaz skrótów obejmuje akronimy rozwijane w tekście (w nawiasach „ang.") oraz standardowe terminy inżynierskie związane z testowaniem systemów wbudowanych.