

**Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie**

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA INFORMATYKI STOSOWANEJ



PRACA MAGISTERSKA

PAWEŁ ENGLERT

**REPOZYTORIUM TESTÓW OPROGRAMOWANIA DLA
PRODUKTÓW WIELOWYDANIOWYCH DEDYKOWANYCH NA
WIELE URZĄDZEŃ**

PROMOTOR:

dr inż. Paweł Skrzyński

Kraków 2 września 2013

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical
Engineering

DEPARTMENT OF APPLIED COMPUTER SCIENCE



MASTER OF SCIENCE THESIS

PAWEŁ ENGLERT

**TEST MANAGEMENT REPOSITORY FOR MULTI-DEVICES
AND MULTI-RELEASES PRODUCTS**

SUPERVISOR:
Paweł Skrzyński Ph.D

Krakow 2 września 2013

Dziękuję Panu dr. inż. Pawłowi Skrzyńskiemu za pomoc, opiekę i rady podczas pisania pracy. Dziękuję również żonie Katarzynie za nieocenione wsparcie.

Spis treści

| | |
|--|----|
| 1. Wstęp | 9 |
| 1.1. Cel pracy | 10 |
| 1.2. Struktura pracy | 10 |
| 2. Wprowadzenie | 11 |
| 2.1. Testowanie w procesie tworzenia oprogramowania | 11 |
| 2.2. Modele tradycyjne wytwarzania oprogramowania | 11 |
| 2.2.1. Model kaskadowy | 11 |
| 2.2.2. Model V | 13 |
| 2.3. Modele iteracyjne wytwarzania oprogramowania | 14 |
| 2.3.1. RAD | 15 |
| 2.3.2. Techniki zwinne | 15 |
| 2.4. Strategie testowania oprogramowania | 16 |
| 2.4.1. Strategie analityczne | 16 |
| 2.4.2. Strategie wynikające z projektu oprogramowania | 18 |
| 2.4.3. Strategie metodyczne | 18 |
| 2.4.4. Strategie zorientowane procesowo | 18 |
| 2.4.5. Strategie dynamiczne | 19 |
| 2.4.6. Strategie sterowane specyfiką testowania | 19 |
| 2.4.7. Strategie regresyjne | 19 |
| 2.5. Typy testów | 20 |
| 2.5.1. Podział ze względu na obszar zastosowania | 20 |
| 2.5.2. Podział ze względu na typ walidacji | 21 |
| 3. Określenie problemu | 23 |
| 3.1. Definicja problemu | 23 |
| 3.2. Rodzaje narzędzi wspomagających proces testowania | 24 |
| 3.3. Potrzeba integracji z innymi narzędziami | 26 |
| 4. Projekt systemu repozytorium | 27 |
| 4.1. Architektura rozwiązania | 27 |

| | | |
|-----------|---|-----------|
| 4.2. | Warstwy systemu | 28 |
| 4.2.1. | Warstwa aplikacji internetowej | 28 |
| 4.2.2. | Warstwa usługi internetowej | 29 |
| 4.3. | Moduły aplikacji..... | 29 |
| 4.3.1. | Moduł administracji | 30 |
| 4.3.2. | Moduł definicji..... | 31 |
| 4.3.3. | Moduł danych uzupełniających..... | 32 |
| 4.3.4. | Moduł wykonania | 33 |
| 4.3.5. | Moduł hurtowni danych | 35 |
| 5. | Implementacja systemu repozytorium..... | 36 |
| 5.1. | Organizacja aplikacji | 36 |
| 5.1.1. | Środowisko uruchomieniowe aplikacji | 36 |
| 5.1.2. | Struktura projektu | 37 |
| 5.2. | Model bazy danych..... | 37 |
| 5.2.1. | Model administracji | 39 |
| 5.2.2. | Model definicji danych dodatkowych | 40 |
| 5.2.3. | Model definicji testów..... | 40 |
| 5.2.4. | Model wykonania testów | 41 |
| 5.3. | Komponenty aplikacji internetowej..... | 42 |
| 5.4. | Implementacja REST..... | 45 |
| 6. | Testy systemu..... | 47 |
| 6.1. | Definicja środowiska testowego | 47 |
| 6.2. | Scenariusz 1: tworzenie bazy testów | 48 |
| 6.3. | Scenariusz 2: tworzenie planu testów..... | 49 |
| 6.4. | Scenariusz 3: wykonanie testu z pozytywnym rezultatem | 50 |
| 6.5. | Scenariusz 4: wykonanie testu z negatywnym rezultatem | 51 |
| 6.6. | Scenariusz 5: Analiza wariacji przypadku testowego | 52 |
| 6.7. | Scenariusz 6: integracja z zewnętrznym oprogramowaniem..... | 53 |
| 6.8. | Podsumowanie scenariuszy testowych | 54 |
| 7. | Zakończenie | 55 |
| 7.1. | Możliwe ścieżki rozwoju aplikacji | 55 |
| 7.2. | Podsumowanie | 56 |

Spis rysunków

| | | |
|-----|---|----|
| 2.1 | Model kaskadowy | 12 |
| 2.2 | Model kaskadowy, podział na części [1] | 12 |
| 2.3 | Model V [2] | 13 |
| 2.4 | Model W [3] | 14 |
| 2.5 | Model <i>butterfly</i> [4] | 14 |
| 4.1 | Architektura systemu | 27 |
| 4.2 | Moduły systemu | 30 |
| 4.3 | Algorytm doboru stanów produktu do przypadku testowego | 34 |
| 5.1 | Organizacja plików projektu | 37 |
| 5.2 | Baza danych, część odpowiedzialna za administrację użytkownikami | 40 |
| 5.3 | Baza danych, część odpowiedzialna za definicje danych dodatkowych | 40 |
| 5.4 | Baza danych, część odpowiedzialna za definicje testów | 41 |
| 5.5 | Baza danych, część odpowiedzialna za wykonanie testów | 42 |
| 5.6 | Strona główna aplikacji internetowej | 44 |
| 6.1 | Stan produktu ze zdefiniowanym warunkiem | 48 |
| 6.2 | Podsumowanie definicji testu | 49 |
| 6.3 | Wybór urządzeń końcowych dla planu testowego | 50 |
| 6.4 | Ekran prezentujący wykonanie testu | 51 |
| 6.5 | Prezentacja zmiany stanu testu | 52 |
| 6.6 | Permutacja dla testu. | 53 |

Spis tablic

| | | |
|-----|---|----|
| 3.1 | Podział narzędzi wspomagających proces testowania oprogramowania według ISTQB (na podstawie [5]) | 26 |
| 4.1 | Metody HTTP/1.1 [6] | 29 |
| 4.2 | Składowe przypadku testowego | 32 |
| 4.3 | Składowe grupy testów | 32 |
| 4.4 | Składowe planu testów | 34 |

1. Wstęp

Testowanie oprogramowania ma na celu wykrycie i poprawienie istniejących błędów, tak by nie występowały one w produkcie końcowym. Celem testowania jest również sprawdzenie czy produkt zachowuje się w sposób oczekiwany przez klienta i spełnia jego scenariusze biznesowe. Założeniem testowania oprogramowania nie jest natomiast przedstawienie dowodu, iż oprogramowanie jest pozbawione błędów. Dowiedzenie bezbłędności oprogramowania jest niewykonalne dla dużych systemów. Teoretycznie istnieje możliwość pełnego przetestowania oprogramowania dla pewnej liczby małych, nieskomplikowanych systemów. Jednak nawet dla małych systemów wykonanie wszystkich możliwych kombinacji przypadków testowych nie jest opłacalne ekonomicznie ze względu na wymagany duży nakład pracy.

Kluczowym zagadnieniem podczas testowania oprogramowania jest więc wykonanie odpowiednich przypadków testowych tak, by przy określonym czasie i wielkości zespołu testującego zapewnić możliwie najwyższą jakość oprogramowania. Istotne jest wykrycie ważnych błędów z perspektywy użycia produktu końcowego.

Cykl testowania oprogramowania można podzielić na kilka faz. Początkową fazą jest analiza wymagań, dokumentacji i innych artefaktów oprogramowania. Wykonywane jest to w celu zebrania istotnych informacji o oczekiwanym zachowaniu. Następnie następuje projektowanie przypadków testowych na podstawie dostarczanych informacji. Kolejnym krokiem jest dobór odpowiednich przypadków testowych do planu testów i ich wykonanie. Podczas wykonania, logowane są wyniki, które są następnie analizowane. Wszelkie incydenty zostają zgłaszane do zespołu programistycznego. Poprawa defektów zostaje weryfikowana, przy czym sprawdzane jest także, czy nie wprowadzono nowych błędów do wcześniej działającego oprogramowania, czynność tą nazywamy regresją.

Cykl poprzez swoją złożoność może być wspierany przez narzędzia informatyczne, specyficzne dla każdej z faz. W ramach niniejszej pracy zostanie zaprojektowane i zaimplementowane repozytorium do przechowywania testów oprogramowania. Poprzez repozytorium autor rozumie centralne miejsce przechowujące wszystkie dane określonego typu, udostępniające prosty sposób przeglądania, edycji i dodawania danych. Repozytorium nie udostępnia dostępu swobodnego, dostęp do repozytorium wymaga autoryzowania poprzez okazanie loginu i hasła użytkownika. Dokładny opis i funkcjonalności stworzonego oprogramowania czytelnik znajdzie w dalszej części pracy.

1.1. Cel pracy

Celem niniejszej pracy jest przedstawienie projektu repozytorium i jego implementacji przy zachowaniu specyficznych funkcjonalności. Repozytorium zapewnia wsparcie dla systemów, których czas życia może być dłuższy niż jedno wydanie i są one dedykowane na wiele urządzeń. Dokładna specyfika zostanie opisana w rozdziale trzecim. Przez system autor rozumie zbiór programów działających w pewnym środowisku, które jako całość dostarczają określonej funkcjonalności i spełniają określone procesy biznesowe.

1.2. Struktura pracy

1. Rozdział pierwszy zawiera wstęp wprowadzający w tematykę pracy i jej cel.
2. Rozdział drugi porusza podstawowe zagadnienia związane z procesem testowania oprogramowania.
3. Rozdział trzeci przedstawia problem poruszony w niniejszej pracy i umieszcza go w rodzinie innych programów wspierający proces testowania.
4. Rozdział czwarty opisuje projekt oprogramowania stworzonego w ramach niniejszej pracy.
5. Rozdział piąty opisuje zagadnienia związane z implementacją oprogramowania powstałego w ramach niniejszej pracy.
6. Rozdział szósty opisuje podstawowe scenariusze użycia i zastosowania stworzonego repozytorium oprogramowania w procesie testowania oprogramowania.
7. Rozdział siódmy przedstawia wnioski i możliwości rozwoju repozytorium.

2. Wprowadzenie

W niniejszym rozdziale przedstawione zostanie zagadnienie testowania oprogramowania w kontekście procesu jego wytwarzania. Omówione zostaną różne rodzaje cykli tworzenia oprogramowania i to w jaki sposób wpisany jest w nie proces testowania. Następnie omówiony zostanie temat strategii testowania oprogramowania i przedstawione ich rodzaje. Na zakończenie rozdziału przedstawione zostaną typy testów w podziale na dwie kategorie: obszar zastosowania i typ walidacji.

2.1. Testowanie w procesie tworzenia oprogramowania

Model tworzenia oprogramowania jest to usystematyzowany proces opisujący jakie kroki (zwane fazami) muszą zostać podjęte w celu stworzenia nowego produktu, bądź nowej wersji produktu. Model determinuje między innymi kolejność faz, ich częstotliwość, czas trwania, możliwość powrotu do faz wcześniejszych. Jedną z faz projektu informatycznego jest faza testowania. W zależności od modelu tworzenia oprogramowania, faza testowania przyjmuje postać całkowicie oddzielnej lub zintegrowanej z innymi wcześniejszymi fazami. Model określa również specyfikę testów, które powinny być wykonane i czas kiedy prowadzone jest projektowanie i analiza testów. Można wydzielić dwa typy modeli tworzenia oprogramowania: tradycyjne i iteracyjne.

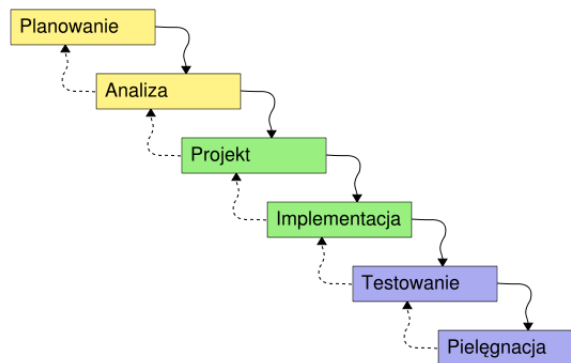
2.2. Modele tradycyjne wytwarzania oprogramowania

Modele tradycyjne zakładają dokładne zdefiniowanie projektu i wykonanie go według liniowo ustalonej kolejności.

2.2.1. Model kaskadowy

Model kaskadowy powstał w celu ujednolicenia faz potrzebnych do stworzenia oprogramowania. Zakłada, iż każda faza następuje po zakończeniu poprzedniej, przy czym przed przejściem do kolejnej fazy nastąpić musi weryfikacja poprzez spełnienie kryterium wyjścia [7]. Model ten zakłada pełną specyfikację wymagań i zaprojektowanie systemu przed implementacją. Pełna definicja wymagań ułatwia zaprojektowanie fazy testowej gdyż dane wejściowe są znane. W modelu tym nie występują błędy związane ze zmianą wymagań podczas implementacji. Z drugiej strony restrykcyjne przestrzeganie pierwotnych założeń powoduje, iż projekt pomimo pozytywnej weryfikacji nie przechodzi fazy walidacji.

Model kaskadowy nie jest więc dedykowany dla projektów informatycznych, które zakładają zmienność wymagań.

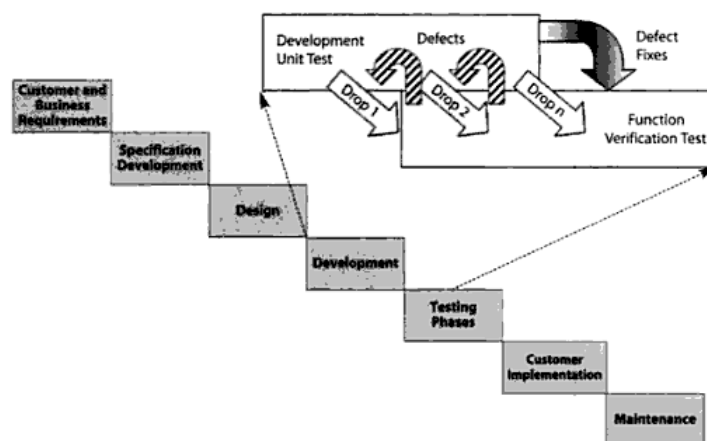


Rysunek 2.1: Model kaskadowy

Można wyróżnić kilka podejść do testowania w modelu kaskadowym. Pierwsze teoretyczne podejście zakłada ścisłe rozdzielenie fazy implementacji od fazy testów, co oznacza, iż nie wykonywane są nawet testy komponentowe. Testy komponentowe jest to rodzaj testów, które operują na poziomie pojedynczych komponentów kodu źródłowego. Drugie podejście zakłada wykonanie testów komponentowych i statycznej weryfikacji podczas fazy implementacji. Statyczna weryfikacja polega na próbie znalezienia błędów i odstępstw od przyjętych norm na podstawie przeglądu kodu źródłowego i innych artefaktów dostarczonych podczas implementacji.

Model kaskadowy stosowany jest najczęściej dla dobrze zdefiniowanych projektów, szczególnie w segmentach bezpieczeństwa publicznego. W takich przypadkach przejście pomiędzy fazami może być połączone z przeglądem i akceptacją formalnych dokumentów.

Rozdzielenie fazy implementacji od fazy testowania powoduje nierównomierną alokację pracowników. Podczas fazy implementacji pracuje zespół programistyczny, który tworzy całą pulę kodu. Zespół ten jest wyłączony z fazy testowania, podczas której pracuje dedykowany zespół testerski.

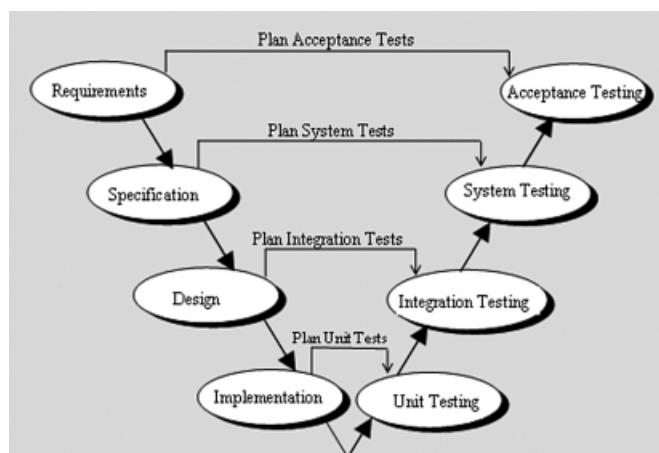


Rysunek 2.2: Model kaskadowy, podział na części [1]

Jedną z wariacji modelu kaskadowego jest rozbitcie tworzonego oprogramowania na części [1]. Zespół programistyczny oddaje pierwszą część do testów. Wykonywane są testy integracyjne i testy systemowe, natomiast znalezione błędy konsultowane są z zespołem programistycznym i zgłaszane. Równolegle zespół programistyczny pracuje nad poprawą zgłoszonych błędów i dokończeniem implementacji części systemu, które nie zostały jeszcze ukończone. Kolejna oddana część jest poddawana testom, weryfikacji poprawionych błędów i małej regresji.

2.2.2. Model V

Model V zakłada rozpoczęcie czynności związanych z planowaniem fazy testów równolegle z fazami analizy, projektu i implementacji. Model obrazuje litera V, dla której lewa część to czynności związane z implementacją i planowaniem, a prawa część to czynności powiązane z testami. Model zakłada, iż każdy typ testu jest połączony z jedną fazą z lewej części modelu. Oznacza to, iż fazy zbierania wymagań, analizy, projektu i implementacji oprócz swoich specyficznych artefaktów dostarczają także analizę wymagań, scenariusze, przegląd dokumentów i kryteria sukcesu do odpowiednich faz testów [2].



Rysunek 2.3: Model V [2]

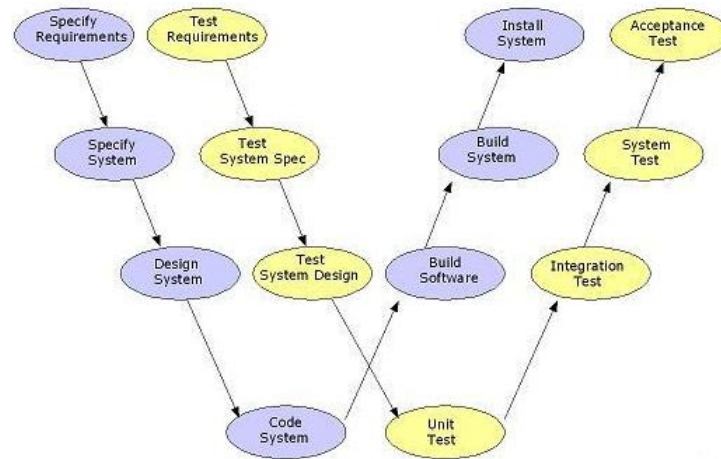
Jedną z głównych zalet modelu V jest to, iż podczas początkowych faz zaangażowany jest zespół testerski, który aktywnie uczestniczy w opisanych wcześniej czynnościach. Wadą modelu jest to, iż rola zespołu testującego zostaje ograniczona do biernego przyjmowania artefaktów bez możliwości ich wstępnej walidacji. Według Rex Black [8], model ten sterowany jest głównie poprzez koszty i harmonogram.

Model ten zakłada, iż kolejność nie jest stała jak w modelu kaskadowym. Każda z faz może spowodować powrót do fazy wcześniejszej, tak więc wymagania mogą ulec zmianie. Zmiana wymagań powoduje konieczność zmiany skryptów do testów.

2.2.2.1. Wariacje modelu V

Praktyczne zastosowanie modelu V powoduje konieczność dostosowania go do aktualnie panujących warunków w organizacji i warunków rynkowych. Jedną z wariacji modelu V jest model W. Model W dostarcza większą władzę zespołowi wykonania testów już w początkowych fazach projektu. Model ten

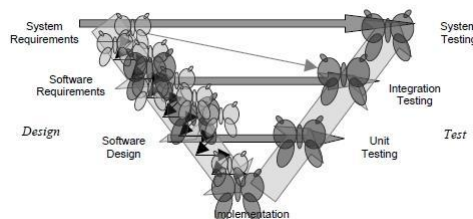
zakłada, iż już podczas fazy analizy i projektowania, dostarczane artefakty są wstępnie weryfikowane i walidowane [3]. Model ten zakłada dla faz z lewej części modelu istnienie równoległych faz, które je kontrolują, weryfikują i walidują. Tak więc dopiero zaakceptowane artefakty służą jako dane wejściowe do procesu planowania odpowiednich faz związanych z testowaniem.



Rysunek 2.4: Model W [3]

Model ten zakłada testowanie projektu we wczesnym stadium. Początkowo wykonywane są testy statyczne i prototypowanie pod kątem użyteczności. Testy dynamiczne wykonywane są gdy zaimplementowane są komponenty.

Rozszerzonym wariantem modelu W, jest model *butterfly*[4]. Model ten zakłada, że każdą z faz można podzielić na kilka mikro-iteracji. Każda z iteracji składa się z analizy pod kątem możliwości przetestowania, projektu testów i ich wykonania.

Rysunek 2.5: Model *butterfly* [4]

2.3. Modele iteracyjne wytwarzania oprogramowania

Modele iteracyjne, w przeciwieństwie do modeli tradycyjnych, zakładają podzielenie projektu na mniejsze części, które są tworzone niezależnie. Można wydzielić dwa typy modeli iteracyjnych:

- ◇ Czysto iteracyjne – co oznacza, iż rozwiązane projektowane jest raz, natomiast faza produkcji i testowania dzielona jest na mniejsze części.

- ◇ Przyrostowe – co oznacza, iż projekt dzielony jest na mniejsze części i każda z części posiada oddzielną fazę projektowania, implementacji i testowania. Każda z części dodaje nowe funkcjonalności.

Modele iteracyjne poprzez podział na podprojekty wymagają od zespołu testującego wykonania regresji począwszy od drugiej iteracji. Regresja ta ma na celu sprawdzenie czy nowo dodany kod nie wprowadził błędów do wcześniej oddanego i przetestowanego rozwiązania.

2.3.1. RAD

RAD czyli szybkie tworzenie oprogramowania (ang. *Rapid Application Development*) to model, który zakłada podział projektu na mniejsze, niezależne moduły, które mogą być implementowane przez odrębne zespoły równolegle. Zespoły w trakcie pracy używają gotowych komponentów i narzędzi do generowania kodu, dostosowując je do indywidualnych potrzeb projektu. Model zakłada, że rozwiązanie może zostać oddane w bardzo krótkim czasie, to jest 30-90 dni roboczych.

Faza testowania zakłada, iż gotowe komponenty używane w projekcie są już przetestowane. Testami należy pokryć dostosowanie rozwiązania pod specyficzną logikę biznesową.

Rozwiązanie to sprawdza się w sytuacji, gdy produkt jest mocno ograniczony czasowo, natomiast jakość i wydajność nie są priorytetem. Wadą rozwiązania jest niska wydajność rozwiązania powodowana używaniem generycznych komponentów. Negatywnie na wydajność wpływa także brak wspólnej pracy między zespołami, które produkując swoje rozwiązanie nie kalibrują się wzajemnie.

Zdarza się, iż oprogramowanie wyprodukowane poprzez ten model jest używane jako prototyp, za pomocą którego projektowane jest końcowe rozwiązanie. Za pomocą modelu *RAD* tworzone jest więc oprogramowanie aż do pewnego momentu, tak by klient mógł skonfrontować swoje przewidywania z działającym oprogramowaniem. W ten sposób walidowana jest poprawność wstępnych wymagań po czym następuje kontynuacja projektu już z zastosowaniem bardziej formalnych technik.

2.3.2. Techniki zwinne

Techniki zwinne zakładają uproszczenie procesu analizy i projektowania oprogramowania przewidując zmienność wymagań w czasie. Głównymi aspektami zwinnych modeli są:

- ◇ Zaangażowanie interesariuszy podczas trwania projektu - model ten przewiduje że conajmniej jeden reprezentant interesariuszy będzie aktywnym członkiem zespołu. Oznacza to, iż zespół projektowy może szybko otrzymać informacje zwrotne.
- ◇ Szybka reakcja na zmieniające się wymagania - w ramach iteracji tworzone są tylko te funkcjonalności, które wchodzi w jej skład. Nie są podejmowane kroki mające przygotować system pod potencjalne funkcjonalności, które często pomimo planów nie wchodzi w zakres produktu.
- ◇ Uproszczenie dokumentacji i wymagań - nie istnieje sformalizowany proces dokumentacji. Niektóre zespoły stosują jedynie dokumentację kodu.
- ◇ Idea wspólnego kodu - każdy członek zespołu ma prawo poprawić kod innej osoby.

- ◇ Duży nacisk na zapewnienie jakości podczas fazy implementacji - stosowane są techniki mające zapewnić wysoką jakość rozwiązania. Jest to np. *TDD*, czyli pisanie testów komponentowych przed rozpoczęciem implementacji.
- ◇ Ciągła integracja i automatyczna regresja - implementacja jest sprzężona z automatycznymi narzędziami do budowania produktu. Oznacza to, iż dodanie nowego kodu do repozytorium powoduje przebudowanie produktu, który może zostać objęty automatyczną regresją bądź manualnymi testami.

Projekt zwinny niesie ze sobą również nowe wyzwania dla zespołu testującego. Pojawiają się nowe techniki, za pomocą których zapewniana jest wysoka jakość oprogramowania. Jednym z przykładów jest programowanie w parach. Technika ta zakłada, iż podczas pisania kodu, programiści pracują we dwójkę zmieniając się, przy czym w danym czasie jedna osoba tworzy kod, natomiast druga kontroluje i szuka lepszych rozwiązań. Forma ta zakłada, iż kod taki jest już wstępnie zweryfikowany i nie wymaga innych formalnych metod.

Ważnym aspektem jest dobra komunikacja w zespole, pomiędzy programistami i zespołem testerskim. W wyniku braku obszernej dokumentacji, pewne informacje przekazywane są bezpośrednio. Rola testera wzbogacona jest o funkcje doradcze, przez co pełni on aktywną rolę już w fazie implementacji.

Zapewnienie automatycznej regresji jest kluczowe dla projektów zwinnych. Musi ona być wykonana po zakończeniu każdej z iteracji, by uzyskać pewność, iż nie wprowadzono błędów do działających rozwiązań.

2.4. Strategie testowania oprogramowania

Projekt informatyczny określa zasady tworzenia i wykonywania testów. Zasady te mają na celu dostarczenie produktu, którego jakość spełnia założone wymagania. Wymagania te są zróżnicowane w zależności od charakterystyki produktu, to jest, systemy medyczne, bankowe czy telekomunikacyjne wymagają krytycznie wysokiej jakości, aplikacje internetowe natomiast cechują się mniej restrykcyjnymi normami. Zbiór reguł i praktyk nazywamy strategią. Strategia testowania oprogramowania determinowana jest głównie przez dwa aspekty: charakterystyka produktu i model tworzenia oprogramowania. Strategia testowania określa sposób tworzenia i wykonywania testów, określa również harmonogram i tryb pracy zespołu testującego.

Typ strategii określa jakie testy będą wykonywane na różnych poziomach testowania. Celem jest stworzenie przypadków użycia i dobór konkretnych skryptów tak, by zapewnić oczekiwany poziom jakości przy minimalizacji kosztów i czasu.

2.4.1. Strategie analityczne

Pierwszą z opisywanych grup strategii jest grupa strategii analitycznych. Zakładają one, iż danymi wejściowymi są artefakty powstałe podczas tworzenia oprogramowania, które następnie poddawane są analizie. Artefakty to na przykład dokumentacja, kod źródłowy, przypadki użycia, lista funkcjonalności.

Poniżej przedstawione zostaną dwie strategie analityczne: strategia sterowana funkcjonalnością i strategia sterowana ryzykiem.

Dla testowania sterowanego funkcjonalnością, jako dane wejściowe używane są funkcjonalności. Projektowanie fazy testowania ma za zadanie pokrycie testami wszystkich wymienionych funkcjonalności. Jest to proces, który złożony jest z dwóch części: walidacji wymagań i identyfikacji przypadków użycia na podstawie wymagań.

Wymagania walidowane są pod kątem wieloznaczności, wzajemnego wykluczania się, niepełnego opisu. Nieprecyzyjne opisy są uzupełniane a dwuznaczności eliminowane. Wyeliminowanie wieloznaczności wymaga obecności interesariuszy, programistów i testerów ponieważ każda z tych grup może interpretować funkcjonalności w inny sposób, co prowadzi do kosztownych błędów w projekcie.

Kolejnym krokiem jest wygenerowanie minimalnej liczby przypadków użycia, które pokryją wszystkie funkcjonalności. Na ich podstawie powstają skrypty testowe. W tym celu tworzony jest diagram przyczyna-efekt. Służy on do zobrazowania wpływu stanu systemu na oczekiwany rezultat dla każdej z funkcjonalności. Po lewej stronie diagramu umieszczane są możliwe warunki wejściowe, po prawej oczekiwany efekt. Pomiędzy dwoma warstwami zachodzą relacje, które mogą posiadać warunki logiczne takie jak: *i*, *lub*, *nie*. Dodatkowo można wyróżnić warunki wejściowe, których odpowiednia wartość powoduje, iż wartość pozostałych elementów nie jest brana pod uwagę, wartości te możemy więc zamaskować. Na podstawie diagramu tworzona jest tabela decyzyjna, która jest źródłem przypadków użycia.

Drugą ze strategii analitycznych jest testowanie sterowane ryzykiem. Strategia ta zakłada, iż najpierw wykonujemy te testy, które dotyczą obszarów oprogramowania mających największe ryzyko. Dobór ryzyka polega na ustaleniu priorytetów dla zdarzeń, które mogą wystąpić i mają negatywny wpływ na jakość oprogramowania.

Ustalenie priorytetów polega na przypisaniu do każdego ze zdarzeń prawdopodobieństwa jego wystąpienia i wpływu jaki może mieć na jakość oprogramowania. Wartości te mogą być liczbowe (np. 1-10), bądź dyskretnie ustalone (np. małe, średnie, duże). Dla każdego możliwego złączenia ryzyko-wystąpienie przypisywana jest waga ryzyka. Ważne jest, aby w trakcie trwania projektu na bieżąco monitorować aktualny stan ryzyka, gdyż może ono zmieniać się w czasie.

Istnieje kilka wyróżnionych domen, do których można przyporządkować poszczególne ryzyka. Spis kategorii pozwala dostrzec pewne powszechne ryzyka, które bez wystarczającej analizy mogą zostać pominięte. Można wyróżnić następujące kategorie ryzyka:

- ◇ funkcjonalność,
- ◇ wydajność,
- ◇ obciążenie,
- ◇ instalacja i deinstalacja,
- ◇ zarządzanie,
- ◇ regresja,

- ◇ użyteczność,
- ◇ jakość danych,
- ◇ obsługa błędów,
- ◇ obsługa daty i czasu,
- ◇ internacjonalizacja,
- ◇ konfiguracja dla różnych środowisk uruchomieniowych,
- ◇ sieci,
- ◇ bezpieczeństwo,
- ◇ dokumentacja.

2.4.2. Strategie wynikające z projektu oprogramowania

Drugim typem strategii są strategie oparte na artefaktach powstałych podczas projektowania oprogramowania. Artefakty takie nazywane są modelami, mogą być to między innymi diagramy przejścia, model domeny, maszyna stanów skończonych. Istnieją programy wspierające tego typu strategię, które generują przypadki użycia bezpośrednio z modeli. Przepływ dla tego typu strategii wygląda następująco:

system → model systemu → skrypty testów → konkretne wykonania testów

2.4.3. Strategie metodyczne

Trzecim typem strategii są strategie metodyczne. Dla tego typu strategii, projekt testów powstaje na podstawie zdefiniowanej metody. Przykładem metody może być metoda uczenia, która polega na stworzeniu listy pomocniczej, która zawiera pytania i zagadnienia, których odpowiedzi mogą dostarczyć istotnej wiedzy przy projektowaniu przypadków testowych. Lista taka tworzona jest na podstawie przeglądu wcześniejszych błędów, wiedzy dziedzinowej i konsultacji eksperckich.

Strategia metodyczna może też korzystać bezpośrednio z metod wynikających ze standardów. Przykładowo standard IBM[9] zakłada podział testowania na kategorie takie jak: użyteczność, funkcjonalności, wersje językowe, dostępność, wydajność, obciążenie, dokumentacja, instalacja.

2.4.4. Strategie zorientowane procesowo

Strategie zorientowane procesowo, są to strategie, których trzonem jest ogólnie przyjęty standard testowania. Przykładem takich strategii może być IEEE 82, czy standardy dla przemysłu lotniczego. Adaptacja strategii wymaga dostosowania ich do specyfikacji produktu. Innym przykładem mogą być opisane strategie testowania zwinnego, które zakładają automatyzację procesu testowania i odporność na zmianę

nawet w późnym etapie projektu. Automatyzacja testowania może zakładać cykliczne wykonywanie grup testów, dla których dane wejściowe są losowe.

2.4.5. Strategie dynamiczne

Dynamiczne strategie testowe zakładają zmniejszony nakład na projektowanie i planowanie fazy testowej. Strategia ta zakłada adaptowanie sposobu testowania do aktualnych warunków. Przypadki testowe tworzone są na bieżąco, przy czym głównie wykonywane są testy eksploracyjne i testy eksperckie. Testerzy wraz z poznawaniem systemu, ustalają priorytety i scenariusze.

2.4.6. Strategie sterowane specyfiką testowania

Strategia sterowana specyfiką testowania oprogramowania zakłada, iż każdy produkt zawiera w sobie błędy. Przyjmowane są z wyprzedzeniem ustalone dolne limity błędów, które może zawierać oprogramowanie. Testowanie prowadzone jest do czasu aż limity zostaną osiągnięte. Oznacza to, iż w podczas testowania oprogramowania dynamicznie dodawane są nowe testy.

2.4.7. Strategie regresyjne

Strategie testów regresyjnych, stosowane są w celu przeglądu, czy nie został wprowadzony błąd w działającej i przetestowanej już funkcjonalności. Z tego powodu stosowane są one często dla produktów posiadających wiele wydań jak i powstających w modelu iteracyjnym. Można wyróżnić trzy rodzaje błędów wykrytych podczas regresji:

- ◇ błąd bezpośrednio wprowadzony przez poprawę defektu lub wprowadzenie nowej funkcjonalności;
- ◇ błąd, który występuje dopiero po naprawie defektu lub dodaniu nowej funkcjonalności;
- ◇ błąd, który pojawił się w innym obszarze produktu lub systemu w związku z nową funkcjonalnością lub poprawą defektu.

Istnieje kilka strategii regresji. Pierwsza strategia zakłada ponowne wykonywanie wszystkich testów wykonanych podczas poprzedniej iteracji lub poprzedniego wydania systemu. Strategia ta związana jest z dużymi kosztami. W celu ich obniżenia stosuje się automatyzację często powtarzanych i długotrwałych testów.

Drugą strategią zakłada wykonanie wybranej puli testów. Dobór testów dokonywany jest na różne sposoby, może to być przykładowo przydział ekspercki polegający na analizie zmian w oprogramowaniu. Alternatywą jest wzmożone testowanie tych elementów, które obarczone są większym ryzykiem bądź tych, które mają krytyczne znaczenie biznesowe lub wpływ na bezpieczeństwo systemu. Powinny zostać również wykonane te testy, które absorbują cały system, tak by potwierdzić że wszystkie elementy współpracują poprawnie.

2.5. Typy testów

Testy oprogramowania można dzielić według różnych kategorii takich jak na przykład: podział ze względu na metodę wykonania testów, podział ze względu na procent automatyzacji, podział ze względu na domenę zastosowania. W niniejszej pracy przedstawiony zostanie podział ze względu na obszar zastosowania i typ walidacji. Wybrano taki podział ponieważ występujące w nim kategorie można spotkać w większości projektów informatycznych.

2.5.1. Podział ze względu na obszar zastosowania

Testy możemy podzielić ze względu na typ składowych oprogramowania, które są weryfikowane i walidowane. Główną charakterystyką, która pozwala wyodrębnić typy jest poziom niezależności i izolacji a także sposób, w który można symulować pozostałą część systemu, która aktualnie nie jest poddawana testom. Wraz z postępem faz projektu wykonywane testy charakteryzują się mniejszym poziomem izolacji i niezależności.

2.5.1.1. Testy komponentowe

Testy komponentowe są to testy, które operują na poziomie pojedynczych klas, modułów kodu źródłowego. Testy te mogą być uruchamiane i testowane niezależnie. Testy komponentowe wykonywane są często w izolacji z innymi częściami systemu. Klasy dostarczające dane, silniki bazodanowe, zastępowane są przez specjalne obiekty, które naśladują ich działanie. Technika ta ma na celu zapewnienie, iż wykrycie błędu podczas testowania określonego modułu nie jest spowodowane błędem wynikającym z błędnych danych pochodzących z modułów zależnych, które nie są aktualnie obiektem testu. Testy takie charakteryzują się wysokim zwrotem inwestycji. Dodatkowo stanowią dokumentację, jako przykład użycia kodu źródłowego.

Testy komponentowe najczęściej wykonywane są podczas fazy implementacji. Wykonywane i tworzone są przez zespół programistyczny, co więcej najczęściej osoba, która tworzy komponent pisze również do niego test. Dobrą praktyką jest, by osoba inna niż autor zweryfikowała czy stworzone testy pokrywają zaimplementowaną funkcjonalność, zdarza się też, iż testy pisane są przed implementacją. Błędy znalezione podczas testów komponentowych najczęściej nie są logowane ponieważ występują przed formalnym oddaniem kodu źródłowego i zatwierdzeniem go.

2.5.1.2. Testy integracyjne

Pojedyncze moduły, dla których testy komponentowe zakończyły się rezultatem pozytywnym są łączone w większe grupy, dla których wykonywane są testy integracyjne, zgodnie z planem testów.

Celem testów integracyjnych jest weryfikacja spełnienia funkcjonalności, niezawodności, wydajności na poziomie większym niż pojedynczy komponent. Testowane są większe grupy logiczne, które dostarczają konkretną funkcjonalność. Główną metodą testowania są testy czarno-skrzynkowe, czyli przeprowadzone testy powstają na podstawie oczekiwanej funkcjonalności, nie na podstawie analizy struktury kodu. Osoby wykonujące testy najczęściej nie posiadają i nie powinny posiadać informacji o wewnętrznym sposobie działania kodu.

Można wyróżnić kilka typów testów integracyjnych, których podział wynika z poziomu izolacji modułów.

Pierwszym typem są testy zależne od całości systemu. Przed przystąpieniem do testowania zakłada się, iż całość systemu jest dostarczona i może zostać zintegrowana. Podczas testowania, używane są prawdziwe implementacje wszystkich potrzebnych modułów. Testowanie tego typu daje pewność, iż system działa poprawnie używając prawdziwych komponentów. Główną wadą jest to, iż testy takie można rozpocząć tylko wtedy gdy gotowy jest cały system, co może nastąpić w późnej fazie procesu tworzenia oprogramowania. Problemem jest także izolacja defektu.

Drugim typem testów, przeciwnym testowaniu całościowemu, jest testowanie polegające na podzieleniu fazy testów integracyjnych na mniejsze fazy, z których każda zakłada testowanie każdej pary modułów. Testowanie tego typu zakłada, iż tylko testowana para musi być realnym oprogramowaniem, reszta systemu może być symulowana. Wymagane jest dostarczenie symulatorów i wspieranie ich podczas kolejnych wydań systemu. Zaletą jest możliwość rozpoczęcia testów już w momencie gdy zespół programistyczny dostarczy gotowy kod dwóch modułów, które ze sobą współpracują. Wydzielenie tylko dwóch modułów pozwala także na wysoką izolację defektów. Wadą jest wysoki koszt i czas trwania tego typu testów gdyż pewne testy powtarzane są dla różnych par modułów.

Pomiędzy dwoma wcześniej opisywanymi podejściami istnieje podejście hybrydowe. Polega ono na łączeniu modułów w grupy niższego poziomu, które są wzajemnie testowane. Następnie grupy niższego poziomu łączone są w grupy wyższego poziomu, które są wzajemnie testowane. Końcowym etapem może być test integracji całego systemu.

2.5.1.3. Testy systemowe

System jest to zbiór zintegrowanych komponentów, które wspólnie realizują wymaganą logikę biznesową. W skład systemu wchodzi także całe środowisko uruchomieniowe, sprzęt, oprogramowanie zewnętrzne. Testowanie systemowe jest określane jako faza testów, które operują na kompletnym w pełni zintegrowanym systemie, działającym na środowisku końcowym lub zbliżonym do końcowego. Testy te sprawdzają zgodność z określonymi wymaganiami takimi jak: funkcjonalności, niezawodność itp. Testy systemowe powinny zostać przeprowadzone po zakończonych testach komponentowych i integracyjnych. Weryfikują one wymagania zarówno funkcjonalne i niefunkcjonalne.

Testy systemowe najczęściej wykonywane są manualnie na podstawie zdefiniowanego planu, przy czym część testów takich jak testy wydajnościowe mogą być wspomagane automatycznie. Testy systemowe zakładają, iż większość negatywnych scenariuszy takich jak podanie błędnych danych sprawdzone zostało podczas faz wcześniejszych testów tak więc testowanie systemowe skupione jest głównie na weryfikacji pozytywnych scenariuszy. Testy systemowe powinny być przeprowadzone przez niezależny zespół, który raportuje do kierownika niezależnego od departamentu produkcji.

2.5.2. Podział ze względu na typ walidacji

Typ walidacji dla testu określa jakiego typu wiedza wymagana jest do jego przeprowadzenia i z jakiej perspektywy tester powinien oceniać oczekiwane rezultaty. Możemy wyróżnić dwa typy walidacji, które skupione są na potwierdzeniu innych charakterystyk systemu.

2.5.2.1. Testy funkcjonalne

Testowanie funkcjonalne ma na zadanie sprawdzić zgodność oprogramowania z zdefiniowanymi wymaganiami. Testy takie przeprowadzane są z punktu widzenia użytkownika końcowego, nie jest więc wymagana wiedza o działaniu i architekturze poszczególnych komponentów. Testy funkcjonalne są zazwyczaj łatwe do testowania ręcznego. Język użyty w opisie testu powinien być dopasowany do terminologii końcowej, tak by móc zweryfikować czy nazwy używane w aplikacji są zgodne z nazwami używanymi w dziedzinie zastosowania.

Przypadek testowy odnoszący się do testu funkcjonalnego powinien zawierać:

- ◇ listę wymagań, które test sprawdza,
- ◇ skrypt testu, czyli listę kroków wraz z oczekiwanymi rezultatami,
- ◇ opis stanu środowiska w jakim należy wykonać test.

Tworząc testy funkcjonalne należy wziąć pod uwagę dwa aspekty: redundancję i strefę szarości. Redundancja testów oznacza, iż podobne testy są powtarzane w różnych fazach. Należy stworzyć taki plan testów aby uniknąć duplikacji. Jeżeli podobne testy występują w różnych fazach należy zadbać by sprawdzały spełnienie wymagań z różnych perspektyw.

Strefa szarości, czyli taka strefa produktu, która nie zostanie pokryta podczas testów. Minimalizować strefę szarości możemy poprzez dobre planowanie. Należy zadbać by całość wymagań została pokrywa przypadkami testowymi. Dodatkowo w planie testów należy zadbać o to by przypadki testowe weryfikujące najważniejsze funkcjonalności dostały większy priorytet (więcej zasobów), natomiast marginalne funkcjonalności mogą być testowane poprzez testy poprzeczne (testujące większą grupę funkcjonalności).

2.5.2.2. Testy нефunkcjonalne

Testy нефunkcjonalne testują jakość oprogramowania. Testowane są нефunkcjonalne właściwości systemu, bez których system, pomimo iż spełnia wymagania nie może zostać nazwany poprawnym. Przypadki testowe dla testów нефunkcjonalnych powinny określać jakościowe i ilościowe oczekiwane rezultaty. Przykładem rezultatu może być określenie "dla 10 000 wejść na stronę system powinien zachowywać się stabilnie co oznacza, iż użytkownicy będą w stanie wykonać swoje procesy biznesowe". Wykonywanie tego typu testów wymaga wiedzy na temat architektury produktu. Powodowane jest to faktem, iż znając newralgiczne części systemu, tester może skupić na nich dodatkową uwagę wiedząc, iż mogą one powodować efekt wąskiego gardła

Testy нефunkcjonalne przez swoją złożoność są trudne lub niemożliwe do wykonania ręcznego. Przykładem może być test wymagający by kilku użytkowników w tym samym momencie załadowało stronę główną portalu internetowego. Jeżeli jest to możliwe, sugerowane jest by testy takie zostały zautomatyzowane.

3. Określenie problemu

W rozdziale zdefiniowany zostanie problem badawczy oraz dokonany zostanie szczegółowy przegląd podobnych problemów i ich rozwiązań. Ostatecznie postanowiona zostanie teza niniejszej pracy.

3.1. Definicja problemu

Tematem pracy jest stworzenie aplikacji wspomagającej proces zapewnienia jakości produktu informatycznego. Istnieje wiele rodzajów aplikacji, które wpisują się w tematykę testowania oprogramowania i wspierają ten proces w różnych fazach i aspektach. Założeniem pracy jest stworzenie aplikacji, która będzie repozytorium testów manualnych. Głównymi funkcjami repozytorium będzie możliwość dodawania i edycji testów, klasyfikowania i definicji, grupowania i wykonywania testów.

Odwołując się do wcześniejszych informacji, aplikacja przeznaczona jest dla każdego z typów wytwarzania oprogramowania. Repozytorium przechowywać będzie testy integracyjne i systemowe, ponieważ te typy testów najczęściej wykonywane są manualnie. Charakterystyka testów może być zarówno funkcjonalna jak i niefunkcjonalna, należy jednak mieć świadomość, iż testy niefunkcjonalne przez swoją złożoność mogą nie być możliwe lub być trudne do wykonania manualnego.

Specyfikacja aplikacji, której projekt i implementacja przedstawione będą w niniejszej pracy umożliwiać będzie definicje testów i planów testowych dla systemów dedykowanych na wiele urządzeń. Aplikacja wspierać będzie iteracyjny model wytwarzania oprogramowania, w którym w ramach jednego wydania systemu przeprowadzonych będzie kilka strategii testowych (dla każdej iteracji osobno). W modelu takim istotne jest to by podzielić testowanie poszczególnych funkcjonalności na inkrementacje, tak by możliwe było jak najszybsze testowanie już oddanych funkcjonalności bez potrzeby wydania całego systemu. Dodatkowym aspektem jest potrzeba przeprowadzenia regresji, tak by uzyskać pewność, iż nowe wydanie systemu nie spowodowało defektów w już istniejących funkcjonalnościach (z poprzednich wersji). Należy również zadbać o regresje między iteracjami, gdyż nowe funkcjonalności mogą wprowadzić defekty w funkcjonalnościach już przetestowanych.

Systemy dedykowane na różne kombinacje urządzeń docelowych lub urządzeń pośrednich posiadają złożoną kombinację możliwych przypadków testowych. Złożoność tą możemy przedstawić wzorem:

$$TC_{mix} = TC * D \quad (3.1)$$

- ◇ TC_{mix} – liczba kombinacji przypadków testowych
- ◇ TC – liczba przypadków testowych
- ◇ D – liczba urządzeń

Powołując się na wiedzę z zagadnień testowania oprogramowania, można stwierdzić, iż niemożliwe jest testowanie wszystkich kombinacji dla przypadków użycia i urządzeń. Celem zapewnienia najwyższej jakości produktu a zarazem zminimalizowania kosztów testów stosowana jest strategia sterowana ryzykiem. Dla przypomnienia polega ona na określeniu, które funkcjonalności objęte są najwyższym ryzykiem w aktualnym wydaniu produktu. Powoduje to, iż testy odnoszące się do najbardziej ryzykownych funkcjonalności otrzymują najwyższy priorytet, tak więc z tych grup zostanie wybrana największa liczba testów, które zostaną wykonane. Analogiczną strategię sterowaną ryzykiem należy zastosować dla urządzeń, dla których produkt jest dedykowany. W tym przypadku pomocna może okazać się znajomość rynku, na który dedykowany jest system, lub konfiguracji jeżeli system dedykowany jest dla jednego klienta. Analiza taka dostarczy dane, które pozwolą skoncentrować proces testowania na kluczowych urządzeniach.

3.2. Rodzaje narzędzi wspomagających proces testowania

Narzędzia wspierające proces testowania możemy dzielić ze względu na różne kategorie takie jak na przykład: cel, poziom testów, dla których są dedykowane, rodzaj licencji, technologia itp. Standard ISTQB stosuje podział w zależności od aktywności, które są wspierane przez narzędzie.

| Rodzina narzędzi | Kategoria | Opis |
|--|---|--|
| Narzędzia wspierające zarządzanie testami (Przykłady: Testlink[10] , Testrail[11] , Fitnesse[12]) | wspierające zarządzanie procesem testowania | przechowują treść testów, plany testów, strategie |
| | wspierające zarządzanie wymaganiami | przechowują priorytety wymagań, zapewniają unikatowość i identyfikację wymagań, wspierają wykrywanie brakujących lub sprzecznych wymagań |
| | przechowujące incydenty | przechowują historię defektów, anomalii, zmian wymagań |
| | zarządzające konfiguracją | przechowują informacje o konfiguracji dla wydań produktów, platform |

Kontynuacja na następnej stronie

Tablica 3.1 – Kontynuacja

| Rodzina narzędzi | Kategoria | Opis |
|---|---|--|
| Narzędzia wspierające testowanie statyczne (Przykłady: Crucible[13], Klocwork[14], Advanced ETL processor[15]) | Wspierające przegląd kodu | przechowują informacje na temat przeglądu, zgłoszone problemy i ich rozwiązania, listy z wskazówkami na temat standardów, udostępniają zdalne tworzenie przeglądów |
| | wykonujące statyczną analizę produktu | generują metryki produktu, sprawdzają zgodność ze standardami, sprawdzają kod produktu pod kątem znanych problemów |
| | modelujące | wspierające walidacje modelu jak na przykład modelu bazy danych, sprawdzają niezgodność relacji, generują przypadki testowe na podstawie modelu |
| Wspierające specyfikacje testów (Przykłady: Generate data[16], Test Composer[17]) | Wspierające projektowanie testów | oferują automatyczne tworzenie przypadków testowych i danych wejściowych na podstawie wymagań, interfejsu użytkownika, kodu |
| | narzędzia przygotowujące dane dla testów | oferują automatyczne wypełnianie systemu danymi do testów (np. generacja danych dla bazy danych) |
| Rodzina narzędzi wspierających wykonanie testów i logowania (Przykłady: Junit[18], Clover[19], Jacoco[20]) | wykonujące testy | wykonują automatycznie lub pół automatycznie testy, zapisują rezultaty |
| | biblioteki testowe | dostarczają komponenty, na podstawie których zespół tworzy testy jednostkowe, symuluje obiekty |
| | porównujące wyniki | sprawdzają zgodność stanu systemu poddanego testów z wymaganiami, pozwalają na określenie czy dany test zakończył się powodzeniem |
| | określające metryki pokrycia | narzędzia określające pokrycie kodu przez testy |
| | narzędzia wspierające testowanie bezpieczeństwa | wspierają system i jego zgodność ze standardami bezpieczeństwa, dostępu do danych, autoryzacji i autentykacji itp |
| Rodzina narzędzi wspierających testowanie wydajności i monitorujących (Przykłady: Jprofiler[21], App Dynamics Lite[22]) | wspierające testowanie wydajności | używane dla testów нефункциональных dla takich dziedzin jak wydajność, obciążenie. Testują wydajność dla dużej ilości wątków, transakcji |
| | wykonujące dynamiczne testowanie | wykonują testy sprawdzające zachowanie systemu podczas jego działania, używane do sprawdzania wycieków pamięci, zależności czasowych |
| | narzędzia do monitorowania | monitorują określone zasoby, pozwalają na analizę porównawczą (na przykład między różnymi wersjami systemu) |

Kontynuacja na następnej stronie

Tablica 3.1 – *Kontynuacja*

| Rodzina narzędzi | Kategoria | Opis |
|--|-------------------------------------|--|
| Rodzina narzędzi specjalnego zastosowania (Przykłady: Universal Replicator[23]) | narzędzia do analizy jakości danych | używane przy testowaniu migracji (aktualizacja oprogramowania do nowej wersji połączona ze zmienioną strukturą danych), monitorują poprawność konwersji danych |

Tablica 3.1: Podział narzędzi wspomagających proces testowania oprogramowania według ISTQB (na podstawie [5])

3.3. Potrzeba integracji z innymi narzędziami

Przedsiębiorstwa informatyzują wiele kluczowych procesów biznesowych. Zdarza się, iż do rozwiązania konkretnego przypadku stosowane jest kilka różnych systemów, pochodzących od jednego wydawcy lub wydawców ze sobą niezwiązanych. Fakt istnienia różnych systemów powoduje konieczność zintegrowania ich wspólnie tak by systemy mogły w sposób zautomatyzowany wymieniać między sobą dane, zdarzenia, komunikaty.

Różnorodność na rynku informatycznym powoduje, iż programy pracujące w tej samej domenie, używane wspólnie do rozwiązania konkretnej potrzeby biznesowej, tworzone są w różnych technologiach, w odmiennych językach programowania. Potrzeba komunikacji pomiędzy programami, które różnią się implementacją i technologiami rozwiązywana jest poprzez zastosowanie standardów, które mogą być używane w sposób interdyscyplinarny.

Poprzedni rozdział zobrazował jak wiele różnorodnych narzędzi wspomagających proces testowania można wyróżnić na rynku. W przypadku oprogramowania wspierającego proces testowania, integracja bywa kluczowym zadaniem. W ten sposób stworzyć można w pełni automatyczne lub pół-automatyczne rozwiązania pokrywające proces zapewnienia jakości oprogramowania.

Repozytorium stworzone w ramach niniejszej pracy wpisuje się w potrzebę integracji z narzędziami zewnętrznymi. Komunikacja udostępniona zostanie poprzez usługę sieciową. Więcej szczegółów znajduje się w rozdziale dotyczącym projektu aplikacji.

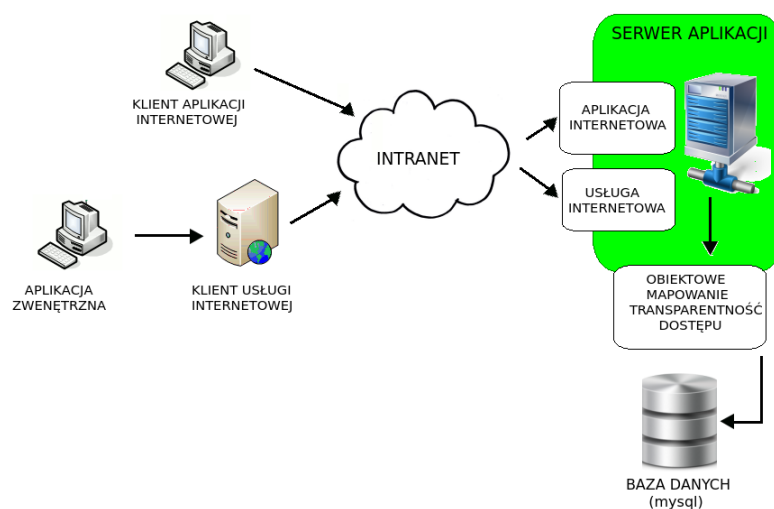
4. Projekt systemu repozytorium

Rodział czwarty opisuje projekt systemu repozytorium testów funkcjonalnych i нефункциональных. Składa się z trzech części. Część pierwsza opisuje ogólną architekturę systemu. Druga część specyfikuje warstwy systemu to jest warstwa aplikacji internetowej i warstwa usługi internetowej. Trzecia część opisuje poszczególne moduły systemu.

4.1. Architektura rozwiązania

Aplikacja stworzona będzie w architekturze klient-serwer. Interfejs użytkownika zrealizowany będzie w technologii aplikacji internetowych, tak by dostęp do repozytorium nie wymagał instalacji i był dostępny na różnych platformach. Dodatkowo stworzona zostanie usługa internetowa (ang. *web service*), tak by umożliwić zintegrowanie aplikacji z zewnętrznymi aplikacjami. Dzięki temu możliwa będzie automatyzacja kluczowych procesów. Utrwalanie danych realizowane będzie w relacyjnej bazie danych poprzez moduł odwzorowujący obiektową architekturę systemu informatycznego na bazę danych.

Aplikacja zostanie napisana w języku JAVA. Język ten jest obecnie najczęściej spotykanym językiem w zagadnieniach korporacyjnych. Posiada rozwinięte wsparcie społeczności i rozbudowane funkcjonalności wbudowane, jak i rozwijane przez zewnętrznych kontrybutorów.



Rysunek 4.1: Architektura systemu

4.2. Warstwy systemu

W projektowanym systemie można wyróżnić dwie warstwy, poprzez które klient może komunikować się w systemie. Pierwsza warstwa to aplikacja internetowa. Przewidziana jest ona do interakcji człowiek-komputer. Druga warstwa, to usługa internetowa, przewidziana jest dla interakcji komputer-komputer. Kolejne dwa podrozdziały opisują specyfikę każdej z warstw.

4.2.1. Warstwa aplikacji internetowej

Dostęp poprzez aplikację internetową jest podstawowym źródłem interakcji użytkownika w projektowanej aplikacji. Celem rozdzielania poszczególnych odpowiedzialności modułów oprogramowania, użyty zostanie wzorzec Model-Widok-Kontroler. Zakłada on wydzielenie trzech warstw:

1. Model - odpowiedzialny za pobranie i enkapsulację danych;
2. Widok - odpowiedzialny za wyświetlenie sformatowanej treści. Język stosowany w widoku powinien pozwalać na swobodne osadzanie treści języka końcowego (w tym przypadku HTML), powinien on być dostosowany do edycji przez osoby nie posiadające wiedzy na temat języku programowania;
3. Kontroler - odpowiedzialny za skoordynowanie pobrania danych, przetworzenia ich za pomocą serwisów i przesłanie danych do widoku.

Język JAVA oferuje technologie objęte zdefiniowanym standardem, które pozwalają tworzyć aplikacje internetowe z wykorzystaniem wzorca Model-Widok-Kontroler. Zaprezentowana aplikacja stworzona zostanie w oparciu o technologię *Java Server Faces*.

Java Server Faces jest jednym ze standardów tworzenia aplikacji internetowych w języku JAVA[24]. Główne założenia standardu to:

1. Łatwość tworzenia części klienckiej (widoku) w oparciu o strukturę komponentową. Udostępnione są standardowe komponenty (takie jak na przykład formularz, pole tekstowe);
2. Możliwość zagnieżdżania struktury dokumentu, pozwalająca na minimalizację redundancji po stronie szablonu strony internetowej;
3. Zdefiniowany standard dostęp z widoku do danych po stronie serwera;
4. Zapewnienie trwałości stanu danych pomiędzy żądaniami w obrębie sesji klienta;
5. Część serwerowa oparta jest na ziarnach (ang. *JavaBeans*), posiada wsparcie dla walidacji zarówno po stronie klienckiej jak i serwerowej.

W technologii *Java Server Faces* rola kontrolera rozdzielona jest na pliki szablonu strony i ziarna zarządzające po stronie serwera. W plikach szablonu osadzone są instrukcje, które wprost pobierają dane z kontrolera i wykonują na nim akcje.

Standard *Java Server Faces* posiada wiele implementacji. W tworzonej aplikacji użyta została implementacja *Mojarra* wzbogacona o komponenty *PrimeFaces*[25]. *PrimeFaces* jest rozwijany jako otwarty projekt. Implementacja ta rozszerza standard o własne komponenty, upraszcza również sposób komunikacji klient serwer poprzez *AJAX* (asynchroniczna komunikacja poprzez język *JavaScript*)

4.2.2. Warstwa usługi internetowej

Jednym ze standardów komunikacji między aplikacjami jest komunikacja poprzez usługę internetową, dla której istnieje wiele technologii. W niniejszej pracy użyta zostanie technologia *REST*. *REST* zakłada iż deklaracja działania, które klient zamierza osiągnąć po stronie serwera jest określona poprzez zasób. Odwołanie do zasobu składa się z:

1. URI czyli adres internetowy określa adres zasobu, do którego odwołuje się klient;
2. typ metody *HTTP* (zgodnie z *HTTP/1.1*) określa jakie działanie ma być podjęte na zasobie (wyświetlenie, dodanie, modyfikacja, usunięcie).

Zgodnie ze specyfikacją *HTTP/1.1* możemy wyróżnić następujące metody i oczekiwany rezultat po stronie serwerowej:

| metoda | mapowanie na akcje |
|----------------------|-------------------------------|
| GET | Wyświetlenie, pobranie zasobu |
| PUT | Stworzenie zasobu |
| POST | Modyfikacja zasobu |
| DELETE | Usunięcie zasobu |
| OPTIONS, TRACE, HEAD | nie używane |

Tablica 4.1: Metody HTTP/1.1 [6]

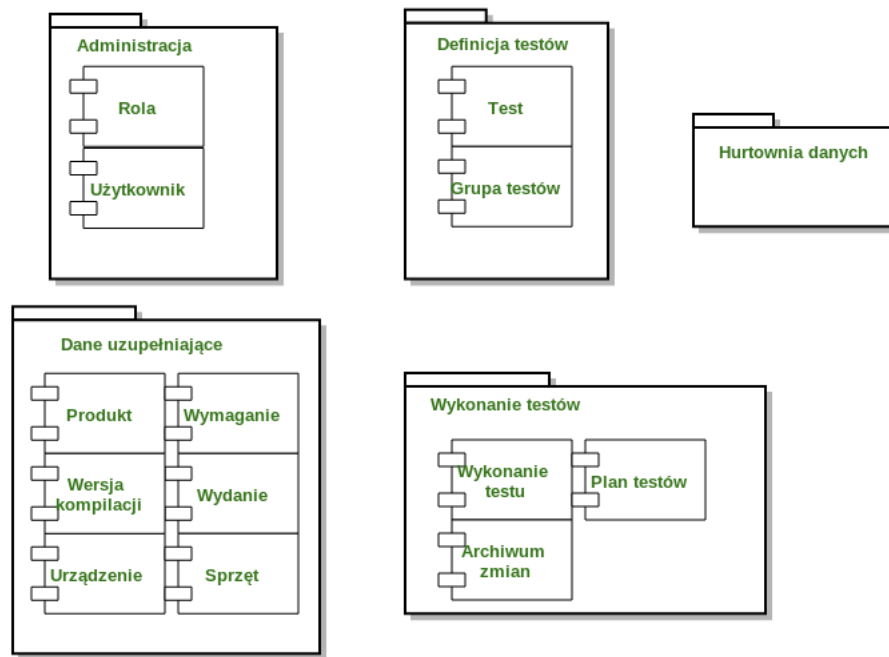
Poprzez usługę internetową możliwe będzie wykonanie następujących akcji:

1. dodanie wymagania;
2. pobrania listy przypadków testowych do wykonania dla użytkownika;
3. aktualizacji wykonania przypadku testowego.

Pierwszy punkt pozwala na integrację z aplikacją przechowującą wymagania. Punkt drugi i trzeci pozwala na integrację z zewnętrznymi aplikacjami do wykonywania testów. W szczególnym przypadku poprzez stworzenie fikcyjnego użytkownika np. automatyzacja, można zintegrować repozytorium z modulem wykonującym testy automatycznie.

4.3. Moduły aplikacji

Funkcjonalności realizowane przez aplikację możemy podzielić na logiczne moduły. Opis poszczególnych modułów czytelnik znajdzie w kolejnych podrozdziałach.



Rysunek 4.2: Moduły systemu

4.3.1. Moduł administracji

Moduł ten skupia wszelkie funkcjonalności związane z zarządzaniem użytkownikami w systemie. Odpowiada za tworzenie i edycję użytkowników. Każdy użytkownik posiada takie dane jak login, adres e-mailowy, hasło jak również przypisaną rolę.

Możemy wyróżnić kilka ról użytkowników. Rola określa jakie uprawnienia otrzymuje zalogowany użytkownik i określa widok ekranu początkowego. Każda z ról posiada charakterystyczne cechy, które pokrywają role użytkowników w procesie testowania oprogramowania: menedżer testów, lider testów, inżynier testów, specjalista środowiska do testów, specjalista konfiguracji testowej [26]. Poniżej zaprezentowany zostanie opis poszczególnych ról.

1. Administrator – zarządza użytkownikami w systemie. Administrator tworzy użytkowników i nadaje im uprawnienia.
2. Koordynator testów – tworzy przypadki testowe, grupy testów i plany testów. Rola ta odpowiedzialna jest za treści merytoryczne repozytorium. Użytkownik odpowiada za utrzymanie testów, aktualizację ich, pokrycie funkcjonalności.
3. Obsługa techniczna – rola ta odpowiedzialna jest za zapewnienie odpowiedniego środowiska dla testerów, konserwację i naprawę fizycznych defektów. Użytkownik ten przypisany jest do konkretnych wykonań scenariuszy, instaluje początkowe środowisko, wymagane wersje oprogramowania, naprawia usterki sprzętowe.

4. Lider testów – wspiera zespół w wykonywaniu planu testów testowego. Służy swoją wiedzą i doświadczeniem przy podziale prac i podczas pojawiających się problemów. Posiada władzę decyzyjną przy zakwalifikowaniu testu jako nie udanego. Lider przypisuje testerów do testów.
5. Tester – wykonuje przypisane do niego przypadki testowe. Odznacza stan testów i zgłasza napotkane problemy.
6. Pośrednik (ang. liaison) – Odpowiedzialny jest za komunikację zespołu testów z zespołem programistycznym. Posiada wgląd do aktualnie wykonywanych testów i konfiguracji. Jego zadaniem jest rozwiązywanie problemów związanych z jego macierzystym produktem. Pomaga zakwalifikować problem powstały podczas testów, szczególnie na początku testów produktu, zespół testerki może nie posiadać wystarczającej wiedzy i błędnie kwalifikować obserwowane rezultaty jako błąd produktu. Pośrednik proponuje również tymczasowe rozwiązania, które pozwalają obejść problemy wynikające z błędów w oprogramowaniu, które naprawione będą dopiero podczas przyszłych wersji oprogramowania, tak by proces testowania mógł przebiegać nieprzerwanie.

4.3.2. Moduł definicji

Moduł ten jest odpowiedzialny za definicję podstawowych jednostek systemu czyli: testów, grup testów i planów testów.

Podstawową jednostką aplikacji jest przypadek testowy. Składowe przypadku testowego przedstawione są w tabeli 4.2;

| Nazwa elementu | Opis |
|-----------------------------|--|
| identyfikator | jednoznacznie identyfikujący test, unikatowy ciąg znaków |
| tytuł | tytuł testu |
| abstrakt | opis testu, jego kluczowe założenia, tematyka i tło określające test |
| grupy urządzeń | Do jednej grupy urządzeń może być przypisane jedno lub więcej urządzeń (w przypadku gdy dana funkcjonalność adresowana jest na więcej niż jedno urządzenie danego typu). Podczas wykonania testu należy jednak określić, którego urządzenia z grupy należy użyć (wybrać jedno) |
| stan wejściowy konfiguracji | Lista sprzętów i ich stanów, które są wymagane do wykonania testu. Dla każdego ze stanów można zdefiniować warunek określający dla jakich konfiguracji grup urządzeń stan ma zajść |
| scenariusz | lista kroków do wykonania wraz z oczekiwanymi rezultatami |
| wymagania | lista wymagań, które są weryfikowane poprzez wykonanie testu |
| estymowany czas | estymowany czas potrzebny do wykonania przypadku testowego |

Kontynuacja na następnej stronie

Tablica 4.2 – Kontynuacja

| Nazwa elementu | Opis |
|---------------------------|--|
| stan początkowy produktów | stan początkowy poszczególnych produktów, który musi być spełniony. Dla każdego ze zdefiniowanych stanów możliwe jest przypisanie warunku, który określa dla jakiej konfiguracji grup urządzeń stan ma zajść |
| liczba wariacji | liczba możliwych alternatywnych przebiegów przypadku testowego (w zależności od doboru urządzeń) |

Tablica 4.2: Składowe przypadku testowego

Przypadki testowe wchodzą w skład grup testów. Hierarchia ta ma drzewiastą strukturę co oznacza iż grupy mogą być zagnieżdżone. Grupy powinny agregować testy, które posiadają podobną charakterystykę. Na przykład testują te same funkcjonalności, wymagają podobnej konfiguracji, urządzeń, dokumentacji. Testy funkcjonalne i niefunkcjonalne nie powinny znajdować się w tej samej grupie. Składowe grupy testów przedstawione są w tabeli 4.3

| Nazwa elementu | Opis |
|-----------------------|--|
| identyfikator | jednoznacznie identyfikujący grupę, unikatowy ciąg znaków |
| identyfikator rodzica | grupy mogą przybierać postać drzewiastą |
| tytuł | tytuł grupy testowej |
| opis | opis grupy testowej, określający jakiego typu testy powinny znaleźć się w grupie |

Tablica 4.3: Składowe grupy testów

4.3.3. Moduł danych uzupełniających

Definicja testów przez swoją złożoność i potrzebę pełnej specyfikacji wymaga pewnych danych dodatkowych, które muszą zostać zdefiniowane w aplikacji. Repozytorium przeznaczone jest dla systemów wiele-wydaniowych i wspierany jest inkrementacyjny model wytwarzania oprogramowania.

Pierwszym krokiem jest zdefiniowanie produktów wchodzących w skład systemu, który poddany zostanie testom. System w kontekście aplikacji jest to zbiór zintegrowanych produktów, które współdziałając oferują określoną funkcjonalność z perspektywy klienta.

W celu wsparcia inkrementacyjnego modelu wytwarzania oprogramowania w aplikacji istnieją takie elementy jak „wydanie produktu” i „wersja produktu”, która wchodzi w skład wydania. Poprzez wersje produktu rozumiany jest konkretny skompilowany stan komponentów systemu. Wersja systemu powinna być jednoznacznie identyfikowalna ponieważ stanowi linię bazową. Na podstawie porównania dwóch poprzedzających się wersji można określić kiedy został wprowadzony błąd regresji. Wydania produktu są to wersje widoczne z poziomu klienta końcowego, w ich skład najczęściej wchodzi wiele wersji.

W ramach wydań produktów definiowane są wymagania. Wymaganie powinno być jasno zdefiniowane tak by możliwe było zweryfikowanie spełnienia wymagania przez oprogramowanie. Na podstawie wymagań tworzone są przypadki testowe.

Drugą charakterystyką aplikacji jest wsparcie dla systemów dedykowanych na wiele urządzeń. Aplikacje umożliwia więc przechowywanie informacji o urządzeniach. Definicja urządzenia powinna zawierać dane podstawowe takie jak nazwa, dostawca, zdjęcie jak i referencję do kompletnej dokumentacji na temat urządzenia. Dostęp do dokumentacji jest kluczowy podczas testów, szczególnie dla młodego stażem zespołu testującego.

W module definicji testów, podczas tworzenie przypadku testowego tworzone są grupy urządzeń. W skład grupy może wejść jedno lub więcej urządzeń, przy czym podczas wykonania przypadku testowego należy wybrać tylko jedno urządzenie dla każdej z grup. Konfiguracja wybranych urządzeń determinuje i modyfikuje końcową treść przypadku testowego. Od tego, które urządzenia zostały wybrane mogą zależeć poszczególne kroki, stany początkowe produktów i wymagany sprzęt. Aplikacja repozytorium udostępnia sposób definiowania warunków dla wcześniej wspomnianych elementów. Warunek określa iż dany element jest ważny (wchodzi w skład definicji przypadku testowego) wtedy i tylko wtedy gdy dla określonej grupy urządzeń wybrane zostało określone urządzenie.

Jednym z elementów wykonania testów funkcjonalnych jest instalacja środowiska do wykonania testów. Instalacja składa się z kilku elementów:

- ◇ dostarczenie i zainstalowanie sprzętu wymaganego do wykonania testów (przykładowo może to być symulator fal radiowych, sejf automatyczny, kabel ethernet, komputer stacjonarny);
- ◇ instalacja platformy do wykonania testów (systemy operacyjne, środowiska uruchomieniowe, modyfikacji systemów, bios, maszyny wirtualne);
- ◇ instalacja oprogramowania w wersji zgodnej z planem testów.

4.3.4. Moduł wykonania

Moduł ten przechowuje informacje o realizowanych planach testowych, testach oczekujących do wykonania i historii wykonywanych przypadków testowych.

Pierwszym krokiem jest utworzenie planu testów. Plan testów określa ramy testów, definiuje co i dlaczego powinno zostać przetestowane, określa konfigurację środowiska testowego. Specyfikacja planu testów znajduje się w tabeli 4.4

| Nazwa elementu | Opis |
|----------------|---|
| identyfikator | jednoznacznie identyfikujący plan, unikatowy ciąg znaków |
| tytuł | tytuł plan |
| opis | opis planu testowego |
| wymagania | lista wymagań, których spełnienie zostanie zweryfikowane podczas przeprowadzania testów |

Kontynuacja na następnej stronie

Tablica 4.4 – Kontynuacja

| Nazwa elementu | Opis |
|----------------|--|
| urządzenia | lista urządzeń dostępnych podczas testów |
| wersja systemu | wersja systemu, która będzie wdrożona podczas testowania |

Tablica 4.4: Składowe planu testów

Po utworzeniu planu testowego należy przypisać do planu testy, które mają zostać wykonane. Wyboru należy dokonać biorąc po uwagę potrzebę spełnienia planu przy określonych zasobach ludzkich, pokrycia wymagań i ograniczeń wynikających z dostępności sprzętu i urządzeń.

Plan testów określa jakie wymagania należy zweryfikować, jest to kluczowy aspekt podczas doboru przypadków testowych. Należy dobrać takie przypadki testowe, które weryfikują potrzebne wymagania. Na tym jednak dobór przypadków testowych nie powinien być zakończony. Należy pamiętać o potrzebie przeprowadzenia regresji. Fakt iż określona wersja systemu dostarcza pewne funkcjonalności nie oznacza iż poprzednio przetestowane funkcjonalności dalej działają poprawnie.

Podczas przypisania przypadku testowego do planu, należy go dookreślić w przypadku gdy posiada grupy urządzeń złożone z więcej niż jednego urządzenia. Oznacza to iż należy wybrać jedno i tylko jedno urządzenie dla każdej z grup. Wybór z jednej strony warunkowany jest dostępnymi zasobami sprzętowymi, a z drugiej strony powinien uwzględniać przetestowanie wszczep systemu dla różnych konfiguracji sprzętowych. Należy unikać w sytuacji, w której nie zostaną wykonane testy na urządzeniach, które są poprawne z perspektywy klienta.

Po dokonaniu pełnej definicji przypadku testowego, rozwiązywane są wszystkie warunki określone w takich elementach jak: stany produktu, kroki testu, warunki początkowe. Dla przypomnienia dla tych elementów można określić warunki, które stanowią iż dany element wchodzi w skład przypadku testowego wtedy i tylko wtedy gdy dostępna jest konkretna konfiguracja sprzętowa (wybrane zostało konkretne urządzenie z grupy urządzeń). Schemat dla stanów produktów można przedstawić pseudokodem znajdującym się w listingu 4.3.

```

1  for stanProduktu in stanyProduktu
2      for warunek in warunkiDlaGrupUrzadzen
3          if wybraneUrzadzenie == preferowaneUrzadzenieDlaStanu
4              DODAJ-STAN-DO-TESTU(stanProduktu)
5
6
```

Rysunek 4.3: Algorytm doboru stanów produktu do przypadku testowego

Rezultat wykonania przypadku testowego może przyjmować jedną wartość z podanych:

- ◇ oczekujący - przypadek testowy oczekuje na przypisanie przez testera (każda osoba z zespołu może się przypisać);
- ◇ przypisany - przypadek testowy przypisany do testera (przypadek nie jest już widoczny w puli);
- ◇ zablokowany - przypadek testowy zablokowany przez błąd w innym przypadku, bądź poprzez niegotową konfigurację;
- ◇ rezultat pozytywny - wszystkie kroki scenariusza wygenerowały oczekiwane rezultaty;
- ◇ rezultat negatywny - przynajmniej jeden z kroków scenariusza wygenerował rezultat negatywny, lub testy eksploracyjne związane z przypadkiem testowym wygenerowały rezultat negatywny;
- ◇ nie możliwy do wykonania - przypadek niemożliwy do wykonania z przyczyny braku zasobów;
- ◇ wymagana analiza - podczas wykonywania przypadku testowego, wyniki poszczególnych kroków scenariusza nie dały jednoznacznej odpowiedzi. Wymagana jest konsultacja lidera i pośrednika (ang. *liaisona*).

Po wykonaniu przypadku testowego, tester oznacza wynik wykonania przypadku testowego i w razie konieczności opisuje wnioski jako komentarz. Tester uzupełnia też realny czas poświęcony na wykonanie przypadku testowego.

4.3.5. Moduł hurtowni danych

Moduł ten odpowiedzialny jest za interpretacje danych istniejących w repozytorium. Przedstawiane są tutaj metryki i wykresy kluczowe z perspektywy lidera, koordynatora i kierownika testów.

Pierwszą z grup raportów jest wykres pokrycia wymagań na poziomie definicji testów. Analiza pozwala określić, które części wymagań nie są pokryte przez przypadki testowe.

Drugą grupą jest pokrycie wymagań na podstawie wykonania przypadków testowych. Pokrycia można analizować odpowiednio z poziomu pojedynczego planu testów, grupy testów i całego wydania systemu. Dane te dostarczają informacji na temat tego, które z wymaganych obszarów zostały pokryte podczas rzeczywistych testów. Dzięki tym informacjom kierownik zespołu testów może zaplanować zakres kolejnych planów testów.

Trzecia grupa dotyczy relacji pomiędzy urządzeniami a przypadkami testowymi. Analiza danych dostarczanych w tej grupie pozwala określić, które urządzenia zostały już przetestowane, a także, które urządzenia należy dostarczyć i włączyć w przyszłych przypadkach testowych.

Czwarta grupa dotyczy procentowego stanu przypadków testowych. Analiza danych dostarczanych przez tą grupę pozwala określić czy wraz z czasem projektu zmniejsza się liczba znajdujących defektów, określić procent testów zablokowanych i inne wnioski, które mogą wpłynąć na zwiększenie efektywności zespołu testującego.

5. Implementacja systemu repozytorium

Rozdział 5 opisuje szczegóły implementacji zaprojektowanego rozwiązania. Rozdział ten podzielony został na cztery części: część pierwsza opisuje ogólną organizację aplikacji, druga część opisuje model bazy danych, trzecia część opisuje implementację części aplikacji internetowej, czwarta część opisuje implementację usługi internetowej.

5.1. Organizacja aplikacji

Sekcja ta opisuje wysoko-poziomą strukturę aplikacji. W podrozdziałach przedstawione zostanie: środowisko uruchomieniowe, na którym zaimplementowana została aplikacja oraz struktura plików i bibliotek w projekcie.

5.1.1. Środowisko uruchomieniowe aplikacji

Platforma korporacyjna języka JAVA (ang. *JAVA Enterprise*) określa sposób tworzenia wielowarstwowych systemów w tym języku. Poszczególne funkcjonalności udostępniane są poprzez komponenty. Przykładowymi funkcjonalnościami oferowanymi przez platformę korporacyjną są: dostęp do bazy danych, serializacja danych do formatu *XML*, zarządzanie sesjami i transakcjami, autoryzacja i autentifikacja, wstrzykiwanie zależności, funkcjonalność aplikacji internetowej, funkcjonalność usługi internetowej. Aplikacje tworzone w oparciu o platformę korporacyjną mogą być wdrażane transparentnie na różnych platformach docelowych. Specyfikacja platformy korporacyjnej JAVA określa również ścieżki i format plików konfiguracyjnych, za pomocą których możliwa jest konfiguracja określonych funkcjonalności.

Środowisko, które obsługuje programy oparte na platformie korporacyjnej nazywamy serwerem aplikacji. Jest to zrąb, który udostępnia usługi i środowisko pozwalające uruchomić i zarządzać aplikacjami w tym aplikacjami internetowymi. Zachowanie serwera aplikacji porównać można do wirtualnej maszyny dla uruchamianych aplikacji, która transparentnie udostępnia określone usługi. Zgodnie ze specyfikacją platformy korporacyjnej serwer aplikacyjny musi udostępniać implementacje standardowych funkcjonalności i serwisów oferowanych przez tę platformę.

Specyfikacja platformy korporacyjnej JAVA określa interfejsy dla komponentów. Serwery aplikacyjne dostarczają natomiast referencyjnych implementacji, przy czym implementacja może rozszerzyć zbiór funkcjonalności o własne funkcjonalności specyficzne dla dostawcy. Podczas życia produktu, istnieje możliwość migracji do innej implementacji serwera aplikacji lub nadpisanie implementacji okre-

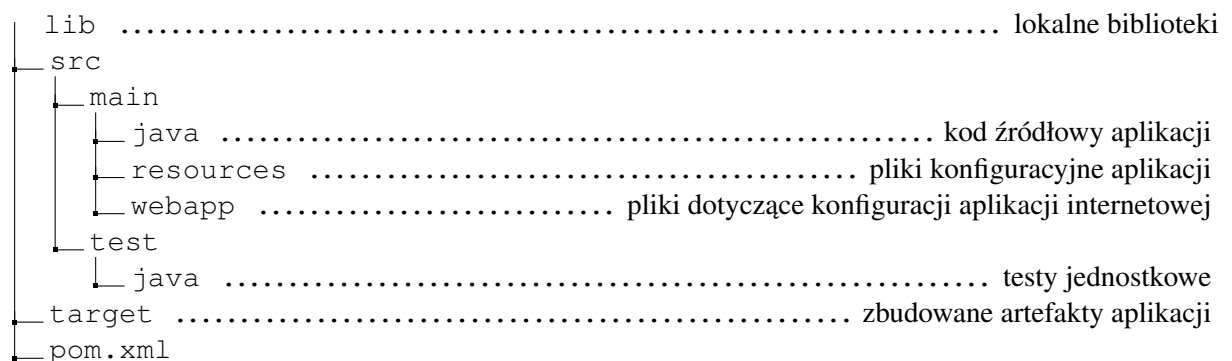
ślonej technologii na implementację innego dostawcy. Tego typu migracje mogą nastąpić transparentnie w przypadku gdy implementacja kliencka korzysta jedynie z funkcjonalności oferowanych przez interfejs. Korzystanie z rozwiązań specyficznych dla dostawcy powoduje konieczność migracji ich na analogiczne rozwiązania dla nowego dostawcy lub stworzenia własnego rozwiązania w przypadku gdy nowy dostawca nie udostępnia wymaganej funkcjonalności.

Istnieje wiele implementacji serwera aplikacji dla platformy korporacyjnej języka JAVA. Podczas tworzenia repozytorium użyty został serwer aplikacji *Glassfish*. Serwer aplikacji *Glassfish*, jest to referencyjna, otwarta i bezpłatna implementacja serwera aplikacji stworzona przez firmę Oracle.

W stworzonej aplikacji użyte zostały jedynie rozwiązania specyfikowane w interfejsach funkcjonalności platformy korporacyjnej. Oznacza to iż migracja na inny serwer aplikacji może odbyć się transparentnie.

5.1.2. Struktura projektu

Pliki źródłowe projektu zostały podzielone na sekcje tak by w sposób przejrzysty rozdzielić poszczególne elementy. Zastosowany podział przedstawiony został na rysunku 5.1. Stworzone zostały osobne katalogi dla: bibliotek, kodu źródłowego i docelowych artefaktów. Pliki kodu źródłowego podzielone zostały na: kod aplikacji i testy jednostkowe.



Rysunek 5.1: Organizacja plików projektu

Wszelkie zależności wymagane przez aplikacje rozwiązywane są poprzez narzędzie do automatycznego budowania projektów *Maven*. *Maven* poprzez konfigurację zdefiniowaną w pliku `pom.xml` określa, które biblioteki są wymagane przez aplikację i pobiera je podczas budowania aplikacji. Narzędzie to poprzez system wtyczek pozwala na zdefiniowanie celów budowania i wdrażania projektu. W zaimplementowanej aplikacji został zdefiniowany cel, który: pobiera biblioteki zależne, kompiluje kod produkcyjny i kod testów jednostkowych, wykonuje wszystkie testy jednostkowe i w przypadku powodzenia instaluje archiwum aplikacji internetowej (ang. *WAR*) na serwerze aplikacji.

5.2. Model bazy danych

Serwer aplikacji *Glassfish* dostarcza bibliotekę *EclipseLink*, która spełnia standardy między innymi warstwy dostępu do danych (ang. *JPA*) i łączenia danych do i z formatu *XML* (*JAXB*). Na poziomie

kodu aplikacji, struktura połączenia obiektów z bazą relacyjną jest rozwiązywana poprzez zastosowanie standardowych adnotacji. Zastosowania standardowych adnotacji zgodnych ze standardem JPA pozwala na transparentność dostawcy modułu mapującego relacyjną bazę danych. W przypadku zmiany dostawcy, migracja dotyczy jedynie pewnych plików konfiguracyjnych, które zawierają specyficzne ustawienia dla dostawcy (np. sposób logowania czy automatycznego tworzenia schematu bazy danych z encji kodu JAVA). Konfiguracja odpowiedzialna za utworzenie połączenia z bazą danych, znajduje się w lokalizacji *src/main/resources/META-INF/persistence.xml*. Poniżej przedstawione główne składowe pliku

◇ `<jta-data-source>java:app/jdbc/repoDataSource</jta-data-source>`

– Referencja do źródła danych

◇ `<property name="eclipselink.logging.level" value="FINE"/>`

– Poziom logowania, parametr specyficzny dla dostawcy modelu orm

Plik ten określa iż aplikacja używa połączenia poprzez źródło danych *java:app/jdbc/repoDataSource*. Źródło danych może zostać zdefiniowane poprzez panel konfiguracyjny serwera aplikacji, lub określone w pliku konfiguracyjnym. W zaimplementowanej aplikacji definicja źródła danych została określona w pliku *src/main/webapp/WEB-INF/glassfish-resources.xml*. W przypadku zmiany serwera aplikacji należy zmienić ten plik na odpowiedni dla dostawcy.

◇ `<jdbc-connection-pool name="java:app/myConnectionPool"
res-type="javax.sql.ConnectionPoolDataSource"
datasource-classname="org.apache.derby.jdbc.ClientDataSource40">`

– Definicja połączenia z bazą danych

◇ `<property name="URL" value="jdbc:derby://localhost:1527/repoTest"/>`

– ustawienia parametrów połączenia z bazą danych

◇ `<jdbc-resource enabled="true"
jndi-name="java:app/jdbc/myDatasource"
object-type="user"
pool-name="java:app/myConnectionPool">
<description />
</jdbc-resource>`

– stworzenia zasobu bazodanowego, do którego można odwołać się w innych plikach związanych z bazą danych

W powyższym pliku konfiguracyjnym zostało zdefiniowane połączenie do bazy danych *java:app/myConnectionPool*. W definicji połączenia zdefiniowane są pola określające sposób połączenia i dane służące do autoryzacji. Poprzez edycje atrybutu *datasource-classname* możliwa jest zmiana dostawcy silnika bazodanowego, tak więc migracja na inny silnik bazodanowy wymaga zmiany danych połączenia i klasy definiującej dostawcę bazy danych. Wdrożenia na serwerze aplikacji może również wymagać instalacji na serwerze aplikacji artefaktów dostarczających kod obsługujący zdefiniowane źródło danych. Następnie definicja połączenia mapowana jest w definicji zasobu bazodanowego, który użyty został w pliku *persistence.xml*.

Zaimplementowany model bazy danych podzielić można na cztery sekcje:

1. administracja,
2. definicja danych dodatkowych,
3. definicja testów,
4. wykonanie testów.

5.2.1. Model administracji

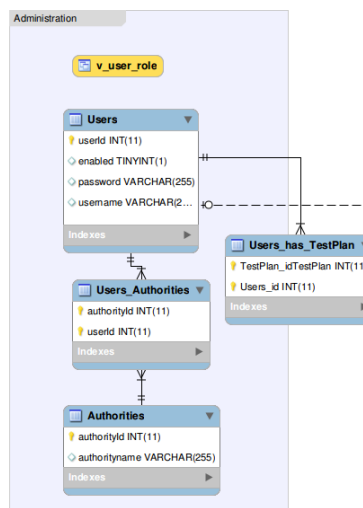
Tabele wchodzące w skład sekcji administracja przechowują dane o użytkownikach i rolach. System posiada predefiniowane role, które określają, które akcje mogą zostać podjęte przez aktualnie zalogowanego użytkownika, rola określa również wygląd głównego menu aplikacji.

Definicja użytkownika składa się z danych identyfikujących go (login w aplikacji, imię, nazwisko), hasła i przypisanej roli, która znajduje się w relacji wiele to wielu z użytkownikiem. Hasła kodowane są w sposób niesymetryczny poprzez algorytm *SHA-256* [27]. Przed zakodowaniem do hasła w postaci tekstowej dodawany jest ciąg zaburzający (ang. *salt*), czyli losowy ciąg znaków o określonej długości. Czynność ta chroni hasła użytkowników przed złamaniem za pomocą tablic tęczowych. Tablice tęcze są to tablice, które zawierają listę przekształceń wejściowych ciągów znaków w ciąg zakodowany. *SHA-256* jest to algorytm haszujący [28], który dla każdego wejściowego ciągu znaków przyporządkowuje 256 bitowy ciąg wyjściowy. Funkcje haszujące są nieodwracalne to znaczy iż nie istnieje przekształcenie, które dla wynikowego ciągu znaków wypisze wejściowy ciąg znaków. Dodatkowymi cechami są:

- ◇ odporność na kolizję – dwa różne ciągi znaków nie generują tych samych wyników;
- ◇ dwa podobne ciągi znaków (podobieństwo może być zmierzone na przykład odległością Levenshteina[29]) generują całkowicie różne ciągi wynikowe.

Autentykacja i autoryzacja w aplikacji realizowana poprzez implementację *JAAS*[30] (ang. *Java Authentication and Authorization Service*) na serwerze aplikacyjnym Glassfish. *JAAS* jest to standard technologii Java, który określa sposób autoryzacji i autentykacji. W celu realizacji standardu *JAAS* należy zdefiniować w pliku konfiguracyjnym role istniejące w systemie i zależności pomiędzy rolami i miejscami w aplikacji, które są dla nich dostępne. Lista dostępowa dla poszczególnych ról, tworzona jest poprzez przypisanie do roli dozwolonych wzorców adresów URL i metod dostępu zgodnych ze standardem HTTP.

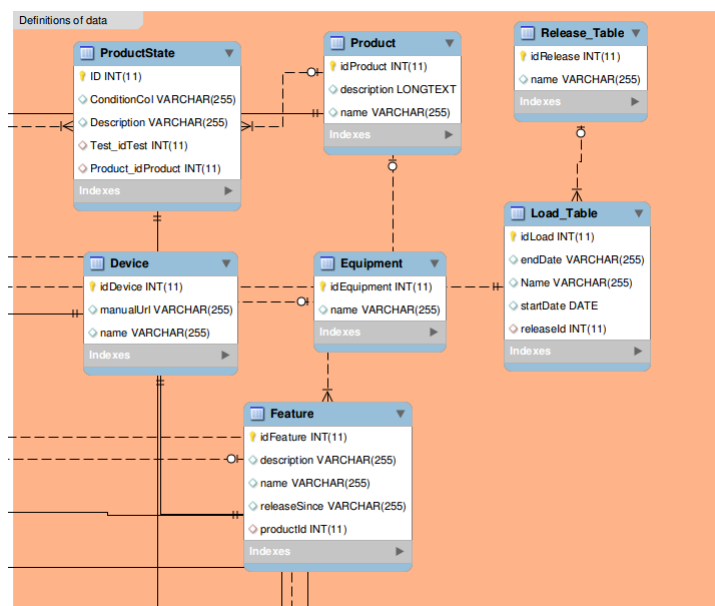
W bazie danych stworzony został widok, który dostarcza zapisane po enkrypcji *SHA-256* hasło i rolę dla podanego loginu użytkownika. Widok ten stworzony został by spełnić wymagania standardu *JAAS*, który oczekuje jednego miejsca dostępowego podczas autoryzacji i autentykacji dla pobrania poprawnego hasła i roli dla użytkownika, który loguje się do systemu.



Rysunek 5.2: Baza danych, część odpowiedzialna za administrację użytkownikami

5.2.2. Model definicji danych dodatkowych

Sekcja ta zawiera encje bazodanowe takie jak: produkt, wydanie produktu, wersja produktu, funkcjonalność, urządzenie, sprzęt. Zdefiniowane encje są łączone w relacji *wiele do wielu* lub *jeden do wielu* podczas tworzenia definicji testu i przy tworzeniu finalnej wersji testu.



Rysunek 5.3: Baza danych, część odpowiedzialna za definicje danych dodatkowych

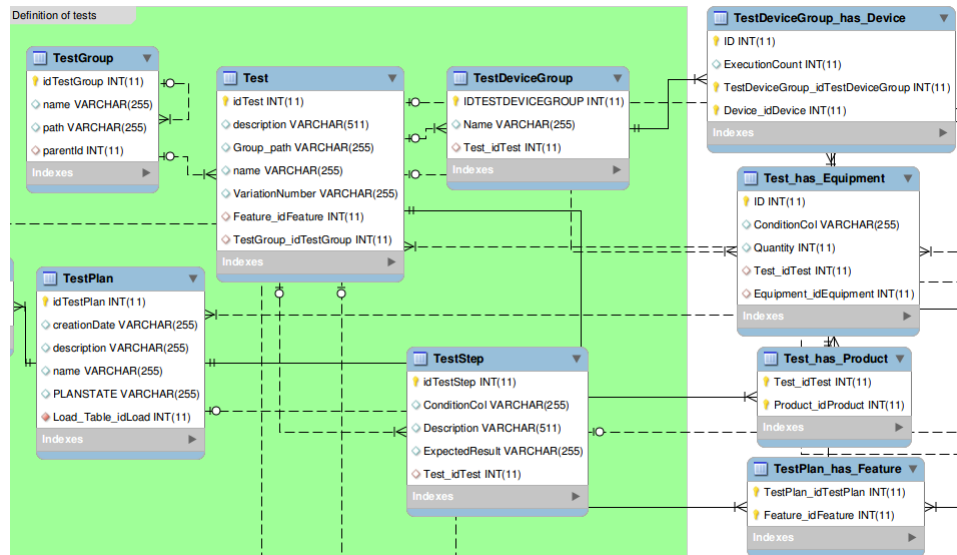
5.2.3. Model definicji testów

Dla tabel Test i TestGroup zastosowano wzorec spłaszczonej ścieżki hierarchii[31] (ang. *materialized path*). Wzorec ten pozwala zoptymalizować pobieranie danych dla struktury drzewiastej w bazie danych. Realizacja polega na przypisaniu dla każdego elementu drzewa, pełnej ścieżki poczynając od

korzenia :

$$idKorzenia.idRodzicaPoziomuN.idRodzicaPoziomu(N-1)...idRodzicaPoziomu1 \quad (5.1)$$

Dzięki takiej strukturze możliwe jest za pomocą jednego zapytania pobranie wszystkich elementów potomnych dla dowolnej ścieżki. Encje bazodanowe będące składowymi testu zawierają warunki lo-

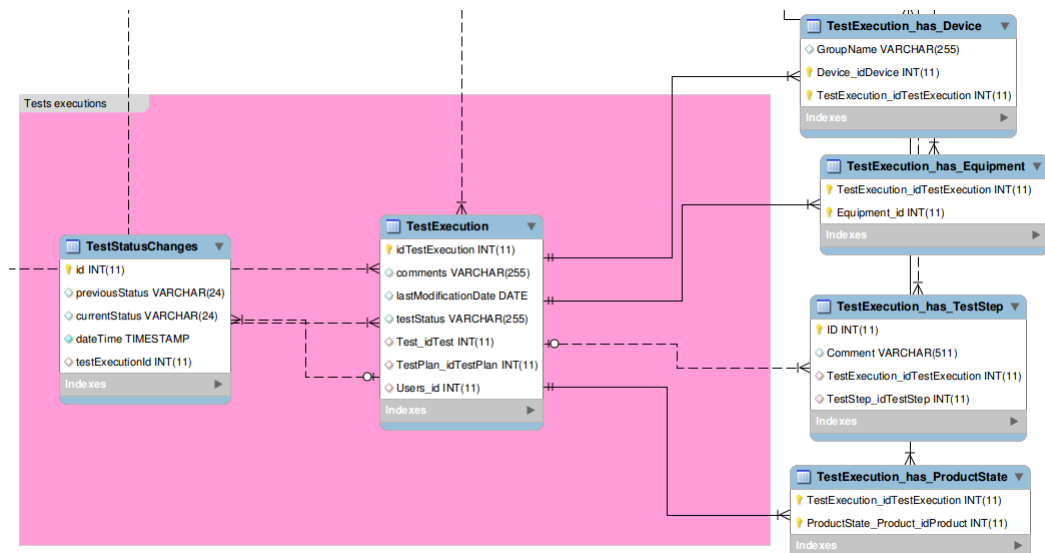


Rysunek 5.4: Baza danych, część odpowiedzialna za definicje testów

giczne stanowiące dla jakich urządzeń dla grup urządzeń składowa wchodzi w skład testu. Normalizacja bazy danych sugeruje iż dla przechowywania tego typu warunku należy stworzyć osobną tabelę, która realizuje relacje wiele do wielu dla urządzeń wchodzące w skład warunku. Rozwiązanie takie spowodowałoby jednak podczas pełnego pobrania encji testu, konieczność wykonania kolejnego złączenia (ang. *JOIN*). Wybrano więc rozwiązanie, które dane dotyczące warunku przechowuje w sposób nieznormalizowany w zamian za większą wydajność. Informacje dotyczące warunku po stronie klienckiej (po stronie kodu Java) przechowywane są w tablicy natomiast podczas utrwalania danych tabela ta jest serializowana i składowana jako tekst w jednej kolumnie. Wzorzec ten zastosowany jest dla encji TesthasEquipment, TestStep i ProductState.

5.2.4. Model wykonania testów

Sekcja ta przechowuje testy wykonane lub oczekujące do wykonania przypisane do planów testowych. Podczas przypisania testu do planu należy określić konfigurację urządzeń, na której test zostanie wykonany. Informacja ta pozwala na rozwiązanie warunków dla encji, których włączenie do definicji testów uwarunkowane jest obecnością określonych urządzeń. Encja wykonania testu zawiera w sobie w relacji jeden do wielu odwołanie do definicji testu. Encje, dla których warunek został rozwiązany pozytywnie przypisane są do wykonania testu poprzez relacje wiele do wielu. Dla relacji pomiędzy wykonaniem testu a pojedynczymi krokami skryptu testowego stworzona została rozszerzona relacja wiele to wielu. Tabela łącząca dla tej relacji zawiera klucze główne dla każdej ze stron relacji i dodatkową kolumnę, która przechowuje treść komentarza.



Rysunek 5.5: Baza danych, część odpowiedzialna za wykonanie testów

Wykonanie testu podczas cyklu życia zostaje przypisane pod konkretnego użytkownika relacją wiele do jednego. Każda zmiana stanu wykonania testu przez użytkownika przechowywana jest w osobnej tabeli archiwizującej, która zawiera historie przejścia między stanami wraz z datą.

5.3. Komponenty aplikacji internetowej

Jak wspomniano w rozdziale dotyczącym projektu aplikacji, warstwa aplikacji internetowej została zrealizowana w technologii *JSF* przy wykorzystaniu implementacji *Mojarra* wraz z komponentami dostarczonymi przez bibliotekę *PrimeFaces*. W celu wdrożenia aplikacji internetowej na serwerze aplikacji należy zdefiniować plik deskryptora *web.xml*. Konfiguracja dla technologii *JSF* znajduje się w pliku *faces-config.xml*, zdefiniowane są tam odwołania do słowników tekstowych, które umożliwiają internacjonalizację aplikacji.

Poniżej przedstawione zostaną składowe pliku *web.xml* związane z konfiguracją aplikacji internetowej pod wymagania technologii *JSF*.

```

◇ <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

```

– definicja serwletu;

```

◇ <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>

```

– Mapowanie żądań klienta do serwletu;

```
◇ <context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>bootstrap</param-value>
</context-param>
```

– Ustawienia dotyczące szablonu aplikacji internetowej.

W powyższym piku konfiguracyjnym określona została ścieżka do implementacji klasy serwletu. Dla technologii *JSF* klasą implementującą serwlet jest *javax.faces.webapp.FacesServlet*. Następnie wszystkie ścieżki zaczynające się od wzorca */facet/* zostały przypisane do wyżej zdefiniowanego serwletu. Oznacza to iż dla wszystkich adresów zaczynających się od wzorca *http://adres-aplikacji.org/faces/**, żądanie zostanie przekazane do serwletu *JSF*, który stworzy i prześle do przeglądarki wyrenderowaną treść strony.

Określony został również szablon prezentacji graficznej aplikacji internetowej. Parametr *primefaces.THEME* określa, który szablon aplikacji powinien zostać załadowany. Użyty został szablon *bootstrap*, jest to szablon, który wyglądem zbliżony jest do bazy szablonów warstwy klienckiej *twitter bootstrap*.

Zdefiniowane zostały akcje możliwe do wykonania przez użytkownika aplikacji internetowej. Dla każdej z akcji zaprojektowane zostały szablony stron, które pozwalają na wykonanie wymienionej akcji. Technologia *JSF* pozwala na kompozycje szablonów. Stworzony został więc główny szablon strony *index.xhtml*, który zawiera w sobie dwie sekcje: **tytuł** i **treść**. Wymienione sekcje rozszerzane są (zapełniane danymi) przez szablony zdefiniowane dla akcji użytkownika.

Plik szablonu dla określonego żądania musi znajdować się na ścieżce wynikającej z adresu żądania. Przykładowo dla żądania *http://adres-aplikacji.org/faces/device/index.xhtml*, wywołany zostanie szablon znajdujący się w lokalizacji *src/main/webapp/device/index.xhtml*.

Kod źródłowy sekcji aplikacji internetowej można podzielić na kilka modułów:

- ◇ klasy znajdujące się w pakiecie **edu.agh.repotest.jsf.controller** – reprezentują kontrolery dla poszczególnych encji, do których dostęp następuje w plikach szablonów;
- ◇ klasy znajdujące się w pakiecie **edu.agh.repotest.converter** – reprezentują klasy konwertujące obiekty języka JAVA na tekst wyświetlany po stronie warstwy HTML;
- ◇ klasy znajdujące się w pakiecie **edu.agh.repotest.session** – reprezentują abstrakcje umożliwiającą operacje na encjach bazodanowych.

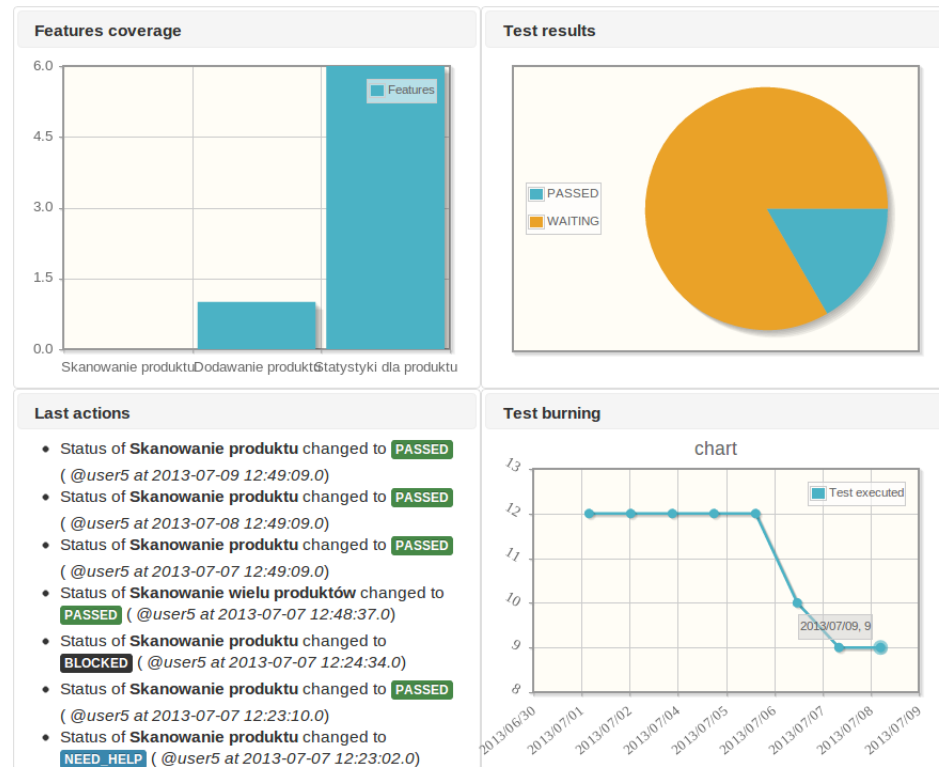
Test Repo for multidevices systems

[Login](#)
[Home](#)

Welcome!

This is main page of the repository. This tool is dedicated to define and track manual tests for the systems which are dedicated for many devices.

To start, lets define some tests and create test plan.



© AGH by Pawel Englert 2013

Rysunek 5.6: Strona główna aplikacji internetowej

5.4. Implementacja REST

Obsługa *REST* zdefiniowana została w pliku deskryptora (*web.xml*). Zdefiniowany został serwlet obsługujący żądania będące żadaniami do zasobów *REST* i ustalone zostały wzorce żądania, które interpretowane są jako żądania *REST*.

```

◇ <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

```

– definicja serwletu obsługującego żądania *REST*;

```

◇ <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>

```

– Mapowanie żądań do zasobów *REST*, wszystkie żądania zaczynające się od wzorca *rest*.

Kod źródłowy języka JAVA zawiera klasy kontrolerów (w pakiecie **edu.agh.repotest.rest**), dla których przypisane są wzorce żądania do zasobu. Mapowanie między adresem zasobu a kontrolerem odbywa się poprzez zastosowanie adnotacji, które określają wzorzec dostępu do zasobu i typ żądania. Na potrzeby usługi internetowej stworzono encje z adnotacjami *JAXB*, które mogą być serializowane do postaci *XML* i *JSON*. Encje te są rodzajem adaptera na encjach bazodanowych.

```

1
2 @Stateless
3 @Path("/users")
4 public class Users {
5
6     @GET
7     @Path("/{username}/testExecutions")
8     @Produces({MediaType.APPLICATION_ATOM_XML, MediaType.APPLICATION_JSON})
9     public List<edu.agh.repotest.dao.TestExecution> showTestsForUser(@PathParam("username") ↵
10         String username) {
11         /*
12         body of the method
13         */
14     }
15 }

```

Listing 5.1: Pobranie testów do wykonania dla użytkownika

Kod źródłowy 5.1 ilustruje sytuację gdy adres żądania *http://adres-aplikacji.org/rest/users/user1/testExecutions* wyświetli listę testów dla użytkownika *user1* w postaci *XML* lub *JSON*. Adnotacja na poziomie klasy określa początek adresu (*users*), adnotacja na poziomie metody określa iż dopasowany zostanie ciąg znaków *./testExecutions*, przy czym pierwszy człon adresu przekazany zostanie do metody jako argument *username*. Adnotacja *GET* określa iż metoda

zostanie wywołana jedynie w przypadku odwołania się poprzez metodę *GET* dla protokołu *HTTP*. Adnotacja *Produces* określa w jakim formacie zwrócone zostaną dane. Końcowy format negocjowany jest z klientem wywołującym żądanie, który przesyła w nagłówku jaki format jest przez niego preferowany.

Poniżej przedstawione zostaną stworzone mapowania dla zasobów:

- ◇ users [GET] – wyświetla wszystkich użytkowników;
- ◇ users/username [GET] – wyświetla informacje o użytkowniku;
- ◇ users/username/testExecutions [GET]– wyświetla testy przypisane dla użytkownika;
- ◇ testExecutions/id [GET] – wyświetla informacje o teście do wykonania;
- ◇ testExecutions/id [POST]– modyfikuje test;
- ◇ features [PUT] – dodaje funkcjonalność.

6. Testy systemu

Rozdział szósty opisuje testy przeprowadzone dla stworzonego systemu repozytorium testów. Celem testów jest weryfikacja poprawnego działania systemu i dowiedzenia spełnienia założonych funkcjonalności. W tym celu zostało określone środowisko testowe na którym wykonano zdefiniowaną pulę testów.

6.1. Definicja środowiska testowego

Weryfikacja aplikacji powstałej jako wynik niniejszej pracy dokonana została poprzez weryfikacje funkcjonalności aplikacji względem przykładowego cyklu życia systemu. Stworzona została charakterystyka systemu, który w pełni może wykorzystać możliwości aplikacji. Następnie stworzone zostały scenariusze, które przeglądowo przechodzą przez możliwe kroki podczas specyfikacji i wykonania fazy testowania oprogramowania.

Specyfikowany system jest systemem zarządzania magazynem o roboczej nazwie *magar*, w którego skład wchodzi następujące produkty:

1. interfejs skanujący kody kreskowe produktów – nazwa robocza **kodex**;
2. część administracyjna do definiowania i dodawania produktów – nazwa robocza **backadm**;
3. część statystyczna prezentująca w sposób graficzny dane zebrane w systemie – nazwa robocza **statos**.

Produkt *kodex* pośredniczy pomiędzy urządzeniami końcowymi czyli skanerami kodów kreskowych a scentralizowaną bazą danych. Założeniem jest możliwość obsługi trzech różnych skanerów kodów kreskowych:

- ◇ skaner1 i skaner2 wyprodukowany przez firmę A,
- ◇ skaner3 wyprodukowany przez firmę B.

Skaner1 i skaner2 posiadają wspólny interfejs konfiguracyjny i podłączane są do systemu za pomocą kabli typu A. Skaner3 posiada natomiast inny sposób konfiguracji i podłączany jest za pomocą kabli typu B.

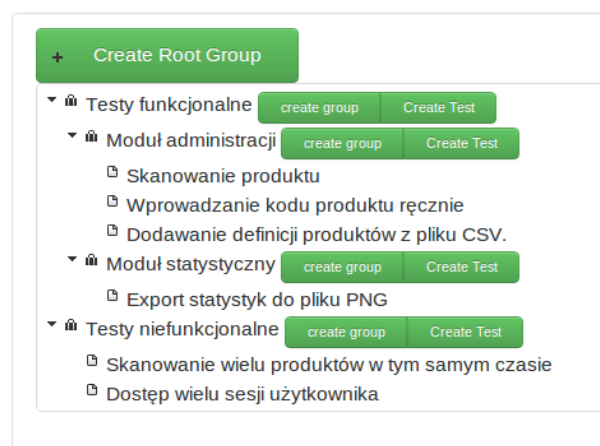
6.2. Scenariusz 1: tworzenie bazy testów

Scenariusz pierwszy polega na stworzeniu trzech różnych grup testów i 6 przypadków testowych. Przynajmniej jeden przypadek testowy powinien korzystać z funkcjonalności tworzenia warunku w zależności od wyboru urządzeń końcowych. W przypadku systemu magur składowe testów warunkowane są użyciem produktów pochodzących od dostawcy A lub od dostawcy B.

Zdecydowano się na utworzenie następującej hierarchii grup testowych (rezultat widoczny po prawej):

- ◇ testy funkcjonalne,
 - część administracyjna,
 - część statystyczna ,
- ◇ testy niefunkcjonalne.

TestGroup Maintenance



Podczas tworzenia definicji testów, sprawdzona została możliwość dodawania elementów, których włączenie w finalną postać testu jest ograniczona spełnieniem warunku. Przykładem takiego elementu jest dodanie warunku początkowego jakim jest konfiguracja skanera przed przystąpieniem do testowania. Skanery pochodzące od różnych producentów posiadają inny interfejs tak więc wszystkie testy wymagające wstępnego ustawienia interfejsu wymagają dodania dwóch rozdzielnych stanów początkowych przy czym tylko jeden ze stanów włączany zostaje do końcowej definicji testu.

Rysunek 6.1: Stan produktu ze zdefiniowanym warunkiem

Po zakończeniu definiowania testu, system wyświetla ekran podsumowujący prezentujący definicje poszczególnych elementów testu wraz z warunkami (Rysunek 6.2).

Test Repo :: for multidevices systems Logged as admin [logout]

Home Administration Test definition Test execution System data Statistics

Creating definition of test

Basic options Devices Product Preconditions Equipments Script Steps **Confirmation**

Moduł administracji / Skanowanie grupy urządzeń

| Name | Value |
|----------|---------------------|
| Features | Skanowanie produktu |
| Products | kodek,backadm |

Groups of devices

| Group name | List of devices |
|------------|-------------------------|
| skanery | skaner1,skaner2,skaner3 |

Test script steps

| # | Instruction | Expected results | Condition |
|---|---|---|-----------|
| 1 | Rozpocząć skanowanie pierwszego produktu | System wyświetla informacje o rozpoczęciu skanowania produktu danego typu | - |
| 2 | Zeskanowanie kolejnych sztuk tego samego produktu | System zwiększa licznik dla produktu | - |
| 3 | Rozpoczęcie skanowania kolejnego produktu | System informuje o rozpoczęciu skanowania nowego produktu | - |

Required products states - preconditions

| Product name | Precondition | Condition |
|--------------|--|---------------------------------|
| backadm | Produkt musi być dodany w bazie danych | - |
| kodek | Należy przekreślić pokrętkę opcji by ustawić opcję multiselekcja na skanerze | skaner1 OR skaner2 FROM skanery |
| kodek | Należy w menu skanera wybrać opcję skanowanie produktów tego samego typu | skaner3 FROM skanery |

Required equipment

| Equipment | Quantity | Condition |
|------------------------|----------|---------------------------------|
| kabel dla producenta A | 1 | skaner1 OR skaner2 FROM skanery |
| kabel dla producenta B | 1 | skaner3 FROM skanery |

[← Back](#)

Rysunek 6.2: Podsumowanie definicji testu

6.3. Scenariusz 2: tworzenie planu testów

Scenariusz drugi zakłada stworzenie dwóch planów testowych, w których skład wchodzi testy tego samego typu. Plan testowy pierwszy zakłada iż testowane są jedynie urządzenia końcowe od dostawcy A, natomiast założeniem drugiego planu testowego jest wykorzystanie urządzeń końcowych pochodzących od dostawcy B.

Stworzenie dwóch planów testowych dla rozłącznych typów urządzeń skutkuje wyprodukowaniem innej końcowej definicji testów do wykonania. Różnice wynikają w doborze dla testów istniejących w poszczególnych planach jedynie tych elementów składowych, które spełniają warunki dla urządzeń końcowych.

Ostatnim krokiem podczas tworzenia nowego planu testów jest określenie urządzeń końcowych dla testów wchodzących w skład testu. Wyboru dokonuje się poprzez wybór dla każdego z testów jednego urządzenia dla każdej ze zdefiniowanych grup urządzeń.

The screenshot shows a web interface titled "Create test plan". At the top, there are three tabs: "Basic", "Select tests", and "Define tests", with "Define tests" being the active tab. Below the tabs, there are two sections for defining devices. The first section is titled "Please define devices for: Skanowanie produktu" and contains a dropdown menu labeled "Select device for Skanery *" with "skaner1" selected. The second section is titled "Please define devices for: Wprowadzanie kodu produktu ręcznie" and contains a dropdown menu labeled "Select device for Skanery *" with "skaner2" selected.

Rysunek 6.3: Wybór urządzeń końcowych dla planu testowego

6.4. Scenariusz 3: wykonanie testu z pozytywnym rezultatem

Scenariusz trzeci zakłada wcześniejsze przypisanie planu testowego dla testera, który następnie wykonuje przypisany dla niego test. Następnie tester powinien mieć możliwość przeglądu przypisanych do niego planów testowych i testów, które oczekują do przypisania.

Kolejnym krokiem jest wykonanie przypadku testowego. Tester powinien móc dodać komentarz dla całego przypadku testowego jak i do poszczególnych składowych skryptu testu. W widoku szczegółów przypadku testowego powinny znajdować się jedynie te składowe, które spełniają określone warunki. Tester powinien móc odznaczyć stan testu jako wykonany pozytywnie. System po zmianie stanu powinien zapisać w archiwum przejście ze stanu ze stanu początkowego do stanu końcowego wraz z datą kiedy to przejście nastąpiło. Przykład przejścia między stanami, którego rezultatem jest rezultat pozytywny obrazuje diagram poniżej.

oczekujący → przypisany → wynik pozytywny

Po przypisaniu testera do przypadku testowego, test ten nie powinien wyświetlać się na liście oczekujących do przypisania dla innych użytkowników. Zmiana stanu na stan końcowy powinna wygenerować zdarzenie w widoku strony głównej aplikacji i zaktualizować wykres spalanie testów.

Ekran prezentujący definicje testu do wykonania dla użytkownika wyświetla końcową definicję testu. Jako iż wykonanie testu posiada już rozwiązane warunki dla poszczególnych urządzeń, wyświetlane są tylko te elementy, które spełniają wymagania dotyczące konfiguracji urządzeń. Ilustracja 6.4 przedstawia ekran wykonania testu, którego aktualny stan to oczekiwanie na wsparcie koncepcyjne gdyż nie można wprost zdiagnozować wyniku testu. W odróżnieniu od ekranu definicji testu, ekran wykonania testu nie przedstawia treści warunków.

Execution of the test

Moduł administracji / Skanowanie produktu

| Name | Value |
|----------|--|
| Status | NEED_HELP |
| Features | Statystyki dla produktu |
| Products | |
| Devices | skaner1 |
| Comment | brak informacji o sposobie podpięcia kabla |

Required equipment for test

| Equipment | Quantity |
|------------------------|----------|
| kabel dla producenta A | 2 |

Required products states - preconditions

| Product name | Precondition |
|--------------|--|
| kodex | Skanowany produkt musi być dodany do bazy danych |

Test script steps

| # | Instruction | Expected results | Comment |
|---|--|--|------------------------------|
| 1 | Podłącz skaner kablem do stacji | kabel podpięty, dioda stacji świeci na zielono | brak |
| 2 | Przyłóż skaner do produktu i naciśnij żółty przycisk | System wyświetla informacje o dodaniu produktu | brak |
| 3 | Sprawdź ilość produktów w systemie | System powinien zwiększyć o jeden ilość dla zeskanowanego produktu | czcionka powinna być większa |

Update test execution and select current status



Rysunek 6.4: Ekran prezentujący wykonanie testu

6.5. Scenariusz 4: wykonanie testu z negatywnym rezultatem

Scenariusz czwarty jest zbliżony do scenariusza trzeciego. Scenariusz ten zakłada iż użytkownik zmieni stan wykonania testu w taki sposób by końcowy stan testu oznaczał negatywny rezultat. Poprzez negatywny rezultat rozumiemy stan gdy przynajmniej jeden z kroków skryptu testowego wygenerował nieoczekiwany rezultat, warunki początkowe nie mogły zostać spełnione, lub obserwacje innego zachowania systemu wskazują błąd w systemie.

Użytkownik powinien mieć możliwość dokonać następującej ścieżki zmianu statusu testu:

oczekujący → przypisany → wymagana analiza → wynik negatywny

oczekujący → przypisany → zablokowany → wynik negatywny

oczekujący → przypisany → wymagana analiza → nie możliwy do wykonania

Zmiany stanu testu powinny być zachowane w archiwum wykonywanego przypadku testowego jak również powinny zostać wyświetlone na stronie głównej aplikacji. Przejście do stanu końcowego powinno zostać uwzględnione na wykresie spalania testów.

Changes history

| Date | From | To |
|-------------------------|-------------|-------------|
| 2013-07-09 19:45:02.85 | NEED_HELP | WAIVED |
| 2013-07-09 19:44:47.615 | IN_PROGRESS | NEED_HELP |
| 2013-07-09 19:43:39.541 | WAITING | IN_PROGRESS |

(a) Archiwum dla wykonania testu

Last actions

- Status of **Export statystyk do pliku PNG** changed to **WAIVED** (@user9 at 2013-07-09 19:45:02.85)
- Status of **Export statystyk do pliku PNG** changed to **NEED_HELP** (@user9 at 2013-07-09 19:44:47.615)
- Status of **Wprowadzanie kodu produktu ręcznie** changed to **PASSED** (@user9 at 2013-07-09 19:43:58.773)
- Status of **Wprowadzanie kodu produktu ręcznie** changed to **IN_PROGRESS** (@user9 at 2013-07-09 19:43:41.369)

(b) Strona główna aplikacji

Rysunek 6.5: Prezentacja zmiany stanu testu

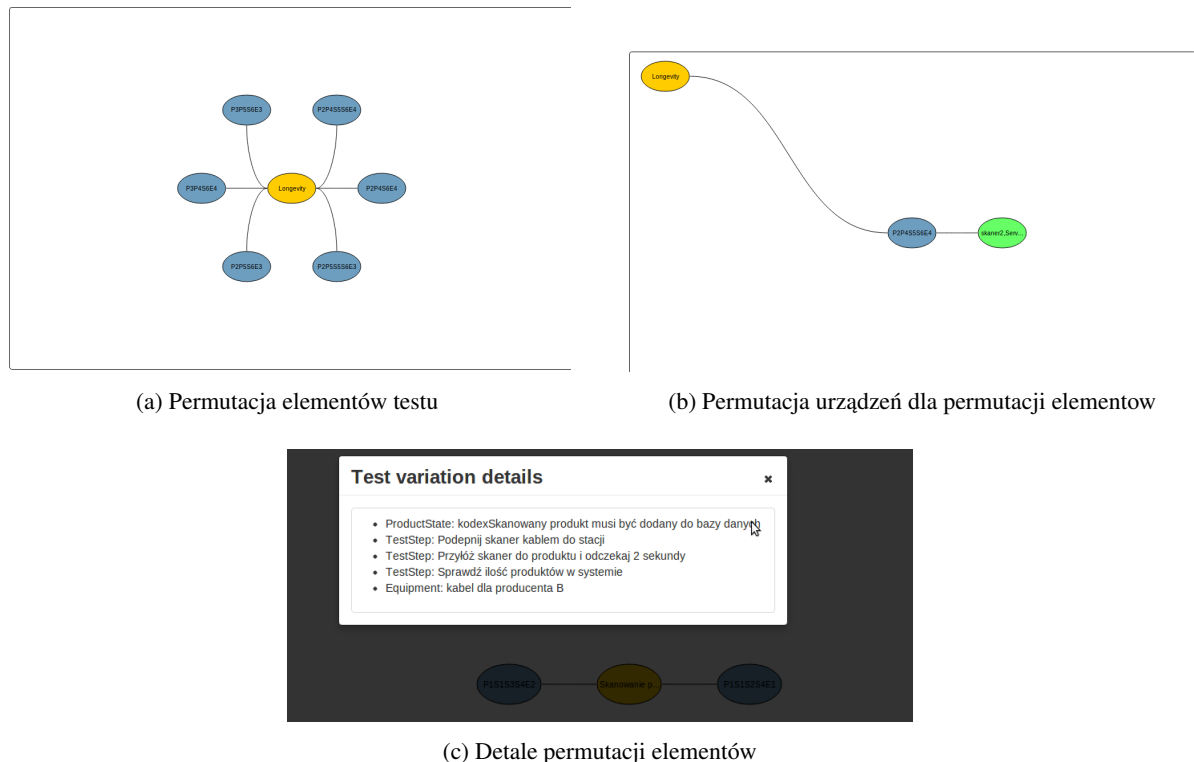
6.6. Scenariusz 5: Analiza wariacji przypadku testowego

Scenariusz piąty weryfikuje funkcjonalności związane z hurtownią danych aplikacji. Typami funkcjonalności oferowanych przez ten moduł są: dane procentowe prezentujące proporcje dla stanów testów, dane prezentujące postęp wykonania planów testów poprzez wykres spalania, dane ilościowe prezentujące pokrycie funkcjonalności przez testy.

Dodatkową funkcjonalnością jest prezentacja możliwych permutacji dla przypadku testowego. Ta funkcjonalność jest weryfikowana bezpośrednio przez scenariusz piąty. Scenariusz ten zakłada iż osoba administrująca systemem jest w stanie przeanalizować możliwe permutacje składowych dla określonego przypadku testowego. System wyświetla informacje na temat permutacji składowych przypadku testowego w postaci grafu skierowanego. W centrum grafu znajduje się przypadek testowy, który połączony jest z wierzchołkami oznaczającymi poszczególne permutacje elementów składowych. Wierzchołki permutacji elementów składowych połączone są z wierzchołkami oznaczającymi permutacje konfiguracji sprzętowej.

Rysunek 6.6 prezentuje widok permutacji dla urządzenia, które posiada definicje dwóch grup testowych. Warunki dla poszczególnych elementów definicji testu zostały zdefiniowane w ten sposób iż dla każdej możliwej permutacji urządzeń z dwóch grup generowana jest inna kombinacja dla końcowej definicji testu. Dla każdej z permutacji elementów końcowych możliwe jest wyświetlenie końcowej treści scenariusza.

Analiza danych dostarczana przez grafy permutacji dla przypadków końcowych może być użyta przez koordynatora testów. Na tej podstawie stworzone mogą zostać plany testowe, które na przestrzeni całego wydania systemu testują wszędy możliwe ścieżki przypadków testowych dla systemu.



Rysunek 6.6: Permutacja dla testu.

6.7. Scenariusz 6: integracja z zewnętrznym oprogramowaniem

Scenariusz szósty weryfikuje możliwości integracji aplikacji z zewnętrznymi aplikacjami istniejącymi już w systemie. Na potrzeby testowania aplikacji użyto aplikacji *POSTMAN rest client*[32], która poprzez interfejs aplikacji internetowej pozwala kierować zapytania typu *REST* do usługi internetowej.

Sprawdzony został możliwy przebieg interakcji komputer-komputer. Założono iż wymaganiem jest możliwość wyświetlania i edycji przypadków testowych dla użytkownika o nazwie *automatyzacja*.

Przypadki testowe dla użytkownika pobierane są poprzez odwołanie się do zasobu konkretnego użytkownika i podzasobu przypadki testowe. Usługa internetowa w zależności od treści nagłówka żądania zwraca odpowiedź w formacie JSON lub XML. Lista przypadków testowych dla użytkownika zawiera odwołania do zasobu oznaczającego poszczególne wykonanie przypadku testowego.

Poprzez odwołanie się do zasobu poszczególnego wykonania przypadku testowego interfejs kliencki komunikujący się z usługą internetową pobiera treść testu do wykonania. Poprzez parsowanie dokumentu XML lub odwołanie się wprost do formatu JSON przez język JavaScript klient wykonuje kroki przypadku testowego. Po zakończonym wykonaniu przypadku testowego, status testu aktualizowany jest poprzez odwołanie się do zasobu poszczególnego wykonania przypadku testowego przy użyciu metody POST.

Aktualizacja przypadku testowego przez usługę internetową wykonuje taki sam przepływ w aplikacji jak wykonanie analogicznej czynności poprzez aplikację internetową.

6.8. Podsumowanie scenariuszy testowych

Wykonanie przedstawionych scenariuszy testowych przebiegło pomyślnie. Aplikacja spełnia założone wymagania funkcjonalne. Ponadto sprawdzone zostały niefunkcjonalne aspekty aplikacji związane z użytecznością. Ocena użyteczności cechuje się subiektywnością gdyż trudno wydzielić jednoznaczne charakterystyki oceniające poziom użyteczności oprogramowania. Aspektami, które mogą pozytywnie wpływać na użyteczność są: przejrzysty i wyraźny dobór kolorów i czcionek w aplikacji, zastosowania odpowiednich komponentów w formularzach, rozbicie dużych formularzy na mniejsze kroki, zastosowanie funkcjonalności przeciągnij i upuść (ang. *drag and drop*).

7. Zakończenie

Rozdział siódmy przedstawia możliwe ścieżki rozwoju aplikacji i podsumowuje rezultat otrzymany w trakcie tworzenia niniejszej pracy.

7.1. Możliwe ścieżki rozwoju aplikacji

Stworzona aplikacja nie jest tworem zamkniętym. Może ona zostać rozwinięta o dodatkowe funkcjonalności tak by spełniać dodatkowe wymagania użytkowników. Można wyróżnić dwa aspekty w kierunku, których nastąpić może rozwój aplikacji:

1. rozszerzenie ogólnych funkcjonalności aplikacji;
2. rozszerzenie wsparcia integracji z zewnętrznymi aplikacjami.

Przykładowymi funkcjonalnościami, które mogą być dodane do aplikacji są: analiza kosztów testowania i system agentowy służący do pobierania danych pochodzących z wykonania testu.

Poprzez analizę kosztów testowania autor rozumie możliwość przypisania określonych kosztów dla poszczególnych elementów składowych testowania. Elementami takimi może być godzina pracy testera, dostarczenie wymaganego sprzętu, koszt instalacji konfiguracji sprzętowej. Zdefiniowane dane mogą posłużyć do analizy kosztów na poziomie permutacji przypadku testowego lub całego planu testów.

Drugą z możliwych funkcjonalności jest zaprojektowanie i zaimplementowanie systemu agentowego, który zbiera artefakty powstałe podczas wykonania przypadku testowego na bieżącym oprogramowaniu i urządzeniach. Agenty te działałyby na urządzeniach, na których zainstalowane są produkty poddane testom. Podczas rozpoczęcia przypadku testowego, system przesyła informacje o rozpoczęciu testu do wszystkich agentów, które znajdują się na komputerach, na których znajduje się oprogramowanie, które wymagane jest do wykonania aktualnego przypadku testowego. W przypadku gdy wynik wykonania przypadku testowego byłby negatywny, system automatycznie pobiera poprzez agenty artefakty, które zostaną dołączone do opisu wykonania przypadku testowego. Artefaktami mogą być: logi aplikacji, stan obciążenia procesora i pamięci. Informacje te mogą być przydatne podczas analizy defektu przez zespół programistyczny.

Drugim kierunkiem rozwoju dla aplikacji jest dodanie dostępu do większej liczby funkcjonalności dla interakcji komputer-komputer. Dobór, które funkcjonalności powinny być również udostępnione przez usługę internetową może zostać dokonany podczas normalnego użytkowania aplikacji. Użytkowanie pozwoli wyodrębnić obszary, które mogą zostać zautomatyzowane. Drugim kryterium doboru

elementów dla usługi internetowej mogą być wymagane wynikające wprost z aktualnego stanu oprogramowania wspierającego proces testowania u użytkownika aplikacji. Pozostałe aplikacje użytkownika mogą przechowywać dane, które w sposób automatyczny powinny zostać przeniesione do repozytorium.

7.2. Podsumowanie

Wynikiem niniejszej pracy magisterkiej jest aplikacja będąca repozytorium testów manualnych. Założeniami było spełnienie specyficznych wymagań dla systemów wielowydaniowych, które dedykowane są na wiele urządzeń. Spełnienie przyjętych założeń walidowane było w rozdziale szóstym, który przedstawia możliwe scenariusze procesu testowania oprogramowania względem aplikacji repozytorium.

Specyfika systemów, dla których dedykowane jest repozytorium wymaga odpowiedniego doboru testów gdyż końcowa ilość przypadków testowych składa się z macierzy istniejących przypadków testowych i permutacji końcowych konfiguracji sprzętowych. Ważnym założeniem aplikacji było umożliwienie jak najprostszego sposobu definicji przypadków testowych, których końcowa treść uzależniona jest od końcowej konfiguracji. Efekt ten udało się osiągnąć poprzez udostępnienie możliwości definiowania warunków dla składowych definicji testów. Drugi aspekt systemów czyli inkrementacyjny sposób tworzenia oprogramowania wspierany jest podczas definicji planów testowych, których specyfikacja wymaga podania wersji oprogramowania. Wersja oprogramowania odpowiada inkrementacji w obrębie jednego wydania.

Aplikacja udostępnia podstawowe funkcjonalności wymagane do zarządzania przypadkami testowymi, planami testów i do wykonywania i raportowania stanu przypadków testowych. Dodatkowo aplikacja udostępnia narzędzia analityczne poprzez hurtownie danych takie jak: wykres spalania testów dla planu testów, wykresy pokrycia funkcjonalności, informacje o zdarzeniach w systemie i grafy permutacji końcowej treści przypadku testowego w zależności od konfiguracji sprzętowej.

Założeniem aplikacji było też umożliwienie integracji z zewnętrznymi aplikacjami. Wymóg ten jest kluczowy ponieważ przedsiębiorstwa na podstawie integracji pojedynczych aplikacji konstruują złożone systemy testowe. Udało się spełnić to wymaganie poprzez udostępnienie poprzez usługę internetową kluczowych funkcjonalności z perspektywy integracji.

Bibliografia

- [1] S. Loveland, M. Shannon, G. Miller, and R. Prewitt, *Software Testing Techniques: Finding the Defects That Matter*. Cengage Learning, 2004.
- [2] J. Watkins and S. Mills, *Testing IT: An Off-the-Shelf Software Testing Process*. Cambridge University Press, 2010.
- [3] P. Herzlich, “The politics of testing.” 1st EuroSTAR conference, London, Oct. 25-28, 1993, 1993.
- [4] S. Morton, “The butterfly model for test development,” 2001.
- [5] ISTQB, “International Software Testing Qualifications Board: Certified Tester Foundation Level Syllabus.” [Online]. Available: <http://www.istqb.org/downloads/finish/16/15.html>
- [6] w3c, “Hypertext Transfer Protocol – HTTP/1.1.” [Online]. Available: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
- [7] S. P. T. B. B. Agarwal, M. Gupta, *Software Engineering and Testing*. United Kingdom: Jones and Bartlett Publishers, 2008.
- [8] R. Black, E. Veenendaal, and D. Graham, *Fundation of Software Testing*. Cengage, 2012.
- [9] R. Abdul, “Effective testing: A customized hand book for testing professionals and students,” 2005.
- [10] Open source, “Testlink website,” accessed: 10-07-2013. [Online]. Available: <http://teamst.org/>
- [11] Gurock Software, “Testrail website,” accessed: 10-07-2013. [Online]. Available: <http://www.gurock.com/testrail/>
- [12] Open source, “Fittesse website,” accessed: 10-07-2013. [Online]. Available: <http://fittesse.org/>
- [13] Atlassian, “Crucible website,” accessed: 10-07-2013. [Online]. Available: <https://www.atlassian.com/software/crucible/overview>
- [14] Klocwork Inc., “Klocwork website,” accessed: 10-07-2013. [Online]. Available: <http://www.klocwork.com/>
- [15] DB Software Laboratory, “Advanced ETL processor website,” accessed: 10-07-2013. [Online]. Available: <http://www.etl-tools.com/etl-tools/advanced-etl-processor/overview.html>

- [16] Open source, “Generate data website,” accessed: 10-07-2013. [Online]. Available: <http://www.generatedata.com/>
- [17] Canam Software Labs, Inc., “Test Composer website,” accessed: 10-07-2013. [Online]. Available: <http://www.canamsoftware.com/Products/CAGenSolutions/TestComposer/Overview/tabid/112/Default.aspx>
- [18] Open source, “Junit website,” accessed: 10-07-2013. [Online]. Available: <http://junit.org/>
- [19] Atlassian, “Clover website,” accessed: 10-07-2013. [Online]. Available: <https://www.atlassian.com/software/clover/overview>
- [20] Mountainminds GmbH Co, “Jacoco website,” accessed: 10-07-2013. [Online]. Available: <http://www.eclemma.org/jacoco/>
- [21] Ej-technologies GmbH, “Jprofiler website,” accessed: 10-07-2013. [Online]. Available: <http://www.ej-technologies.com/products/jprofiler/overview.htm>
- [22] AppDynamics, “AppDynamicsLite website,” accessed: 10-07-2013. [Online]. Available: <http://info.appdynamics.com/AppDynamicsLiteJavaNetOLD.html>
- [23] Hitachi Data Systems Corporation, “Universal Replicator website,” accessed: 10-07-2013. [Online]. Available: <http://www.hds.com/products/storage-software/universal-replicator.html>
- [24] R. Davies and A. Davis, *JavaServer Faces 2.0: The Complete Reference*. McGraw-Hill Osborne Media, 2009.
- [25] O. Varaksin and M. Caliskan, *JavaServer Faces 2.0: The Complete Reference*. Packt Publishing, 2013.
- [26] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.
- [27] N. Sklavos and O. Koufopavlou, “On the hardware implementations of the sha-2(256,384,512) hash functions.” *Proceedings of the IEEE International Symposium on Circuits and Systems ISCAS* 2003, 2003.
- [28] —, “Nist. announcing the secure hash standard.” *Federal Information Processing Standards Publication 180-2*, 2002.
- [29] A. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” 1966.
- [30] SUN, “Java authentication and authorization service (jaas).” [Online]. Available: <http://java.sun.com/products/jaas/>
- [31] H. Kuno and E. Rundensteiner, “Augmented inherited multiindex structure for maintenance of materialized path query views,” 1996.
- [32] A. Asthana, “<http://www.getpostman.com/>.” [Online]. Available: <http://www.getpostman.com/>