POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Marcin Piskor
Nr albumu s9821

Systemy informatyczne w analizie efektywności linii produkcyjnych

Praca magisterska napisana
pod kierunkiem:

dr inż. Mariusza Trzaski

Warszawa, wrzesień 2013

Streszczenie

W poniższej pracy przedstawiono problematykę związaną z właściwym sposobem interpretowania i analizowania danych produkcyjnych. Wyznaczany za ich pomocą współczynnik pozwala na prawidłową ocenę pracy urządzenia produkcyjnego. Jednak bardzo często procesy produkcyjne realizowane są przy udziale wielu maszyn tworzących tzw. linie produkcyjne. Wyznaczenie współczynnika efektywności poszczególnych jej elementów tylko w pewnym zakresie pozwala na ocenę całości procesu produkcyjnego. Zależności występujące pomiędzy urządzeniami mogą w znacznym stopniu wpłynąć na odpowiednią ocenę procesu produkcyjnego. Istniejące obecnie rozwiązania systemowe w różny sposób podchodzą do opisywanej problematyki. Większość z nich umożliwia analizowanie efektywności jedynie dla poszczególnych maszyn wchodzących w skład linii produkcyjnej. Istnieją również rozwiązania pozwalające na wyznaczenie współczynnika efektywności dla całej linii produkcyjnej. Najczęściej jednak, systemy takie implementowane są wyłącznie dla określonej charakterystyki linii lub wymagają stosowania dodatkowych urządzeń monitorujących, co wiąże się z dużymi kosztami wdrożeniowymi.

W przygotowanej pracy zaprezentowano narzędzia i technologie umożliwiające stworzenie generycznego systemu informatycznego analizującego współczynnik efektywności z możliwością przystosowania go do każdego rodzaju linii produkcyjnej. Zaprojektowany i opisany prototyp wyposażono w funkcjonalność, która umożliwia jego współpracę z zewnętrznymi modułami. Każdy z takich modułów, przeznaczony dla wybranej linii produkcyjnej, stanowi indywidualną definicję zależności, jakie zachodzą pomiędzy urządzeniami biorącymi udział w danym procesie produkcyjnym. Korzystając dodatkowo z odpowiednich danych produkcyjnych umożliwiono w ten sposób przeprowadzenie analizy efektywności dla dowolnie wybranej linii produkcyjnej. Stworzony prototyp systemu oparto o technologię desktopową współpracującą z systemem bazodanowym mySQL. Tworzone dla poszczególnych linii produkcyjnych moduły, zaimplementowano w postaci zewnętrznych wtyczek (*plugins*) umożliwiając stworzonemu systemowi odpowiednie ich zarządzanie i wykorzystanie.

Spis treści

1. WSTĘP 4

1.1. Cel pracy 6
1.2. Rozwiązanie przyjęte w pracy 6
1.3. Rezultaty pracy 6
1.4. Organizacja pracy 6

2. PRZEGLĄD ISTNIEJĄCYCH ROZWIĄZAŃ 8

2.1. GOLEM OEE SuperVisor 8
2.2. Computerised Maintenance Management System firmy ESS Ltd. 10
2.3. Provideam OEE 12
2.4. System SMLP 15
2.5. Podsumowanie 16

3. ZAŁOŻENIA PROPONOWANEGO ROZWIĄZANIA 18

3.1. Program główny 18
3.2. Modułowość projektowanego system 19
3.3. Analiza danych 19

4. ZASTOSOWANE NARZĘDZIA I TECHNOLOGIE 22

4.1. Język Java 22
4.2. Biblioteka Swing 24
4.3. MigLayout 26
4.4. Java Simple Plugin Framework 28
4.5. JFreeChart 30
4.6. Eclipse IDE 32
4.7. Baza danych mySQL 35
4.8. Hibernate 37

5. IMPLEMENTACJA SYSTEMU 42

5.1. Program główny 42
5.2. Mapowanie obiektowo-relacyjne 43
5.3. System pluginów 50
5.4. Prezentacja danych 53

6. PODSUMOWANIE 55

7. BIBLIOGRAFIA 56

8. SPIS RYSUNKÓW 57

9. SPIS TABEL 58

10. SPIS LISTINGÓW 59

1. Wstęp

Poziom efektywności jest jednym z najważniejszych zagadnień dotyczących procesu produkcyjnego. Firmy produkcyjne kładą duży nacisk na jak najlepsze wykorzystanie linii produkcyjnych oraz zmniejszenie tolerancji dla strat wynikających z procesu produkcyjnego. Wdrożenie odpowiedniego systemu informatycznego, którego celem miałaby być kontrola i analiza linii produkcyjnych znacznie ułatwiłoby wprowadzenie zmian, by skutecznie podnieść operatywność danej linii.

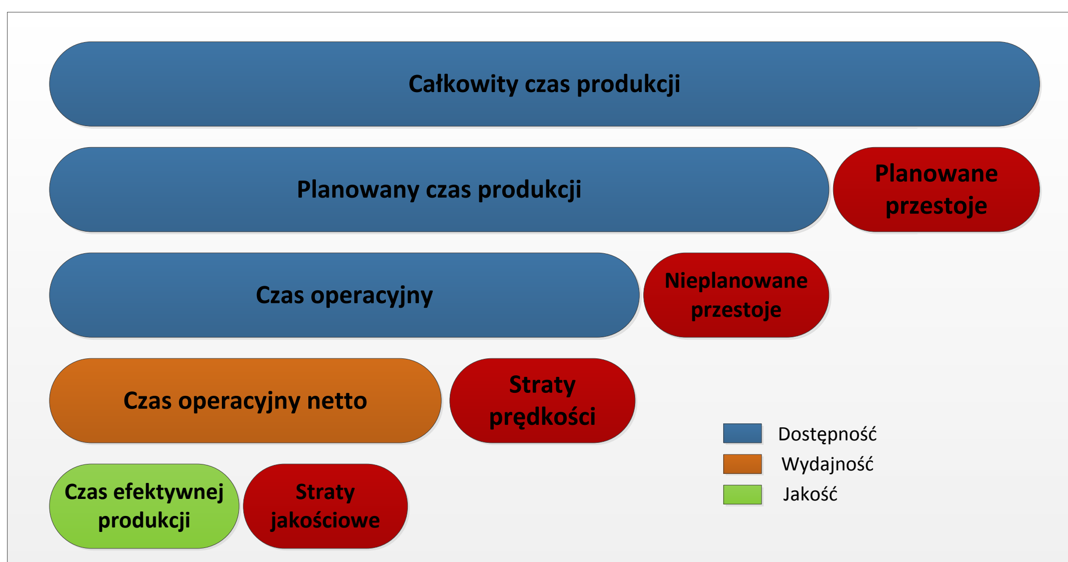
Do określenia efektywności danego urządzenia używany jest wskaźnik OEE (*Overall Equipment Effectiveness*), który jest wypadkową trzech innych wskaźników (patrz [1],[16]):

- **dostępność** (*Availability*) – wielkość wyrażona procentowo, określająca stosunek czasu operacyjnego do czasu planowanego czasu produkcji,
- **wydajność** (*Performance*) – wielkość wyrażona procentowo, określająca stosunek czasu operacyjnego netto do czasu operacyjnego,
- **jakość** (*Quality*) – wielkość wyrażona procentowo, określająca stosunek czasu efektywnej produkcji do czasu operacyjnego netto.

$$OEE = A \times P \times Q$$

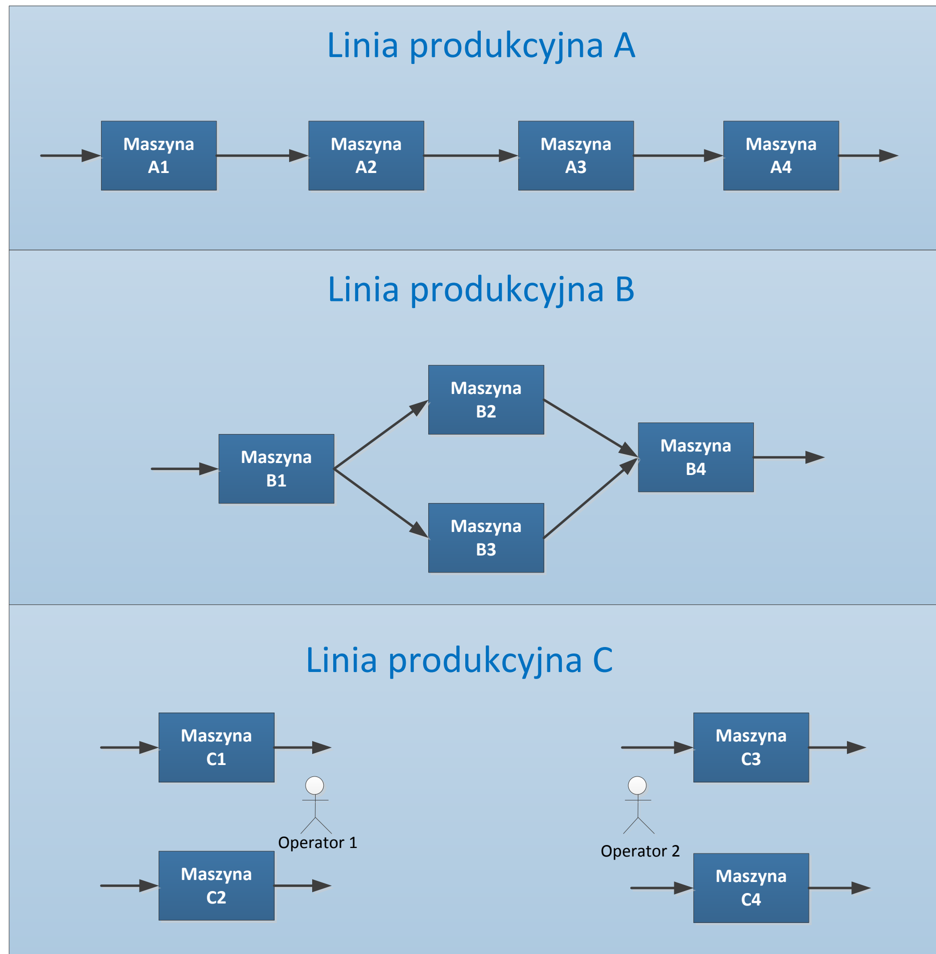
, gdzie:
OEE – wskaźnik efektywności,
A – Dostępność,
P – Wydajność,
Q – Jakość.

Na rysunku 1 przedstawiono graficzną reprezentację wskaźnika OEE z wyszczególnionymi trzema składowymi elementami wpływającymi na jego wartość.



Rysunek 1 Graficzna reprezentacja wskaźnika OEE. Źródło: Opracowanie własne.

Wyznaczenie wskaźnika OEE dla określonego urządzenia nie jest problematyczne, na rynku istnieje szereg rozwiązań oferujących systemy umożliwiające takie obliczenia. Jeśli natomiast chcemy wyznaczyć poziom efektywności dla grupy urządzeń pracujących jako linia produkcyjna, niezbędna jest wiedza na temat jej konstrukcji i potraktować ją, jako jedno duże urządzenie, w którego składzie znajdują się inne maszyny produkcyjne.



Rysunek 2 Przykład organizacji linii produkcyjnych. Źródło: Opracowanie własne.

W zależności od przeznaczenia konkretnej maszyny można oszacować, jak jej obniżona efektywność wpływa na efektywność całej linii. Na rysunku 2 zobrazowano trzy przykłady organizacji linii produkcyjnej. Dla linii produkcyjnej A współczynnik efektywności OEE jest taki, jak najniższy z pośród współczynników efektywności maszyn wchodzących w jej skład. Jeśli awarii ulegnie maszyna A3, to cała linia produkcyjna wytwarza mniejszą ilość produktu, a więc jej efektywność jest zależna w takim samym stopniu od każdej z tworzących ją maszyn. W przypadku linii produkcyjnej B awaria, przestój lub spowolnienie maszyny B1 lub B4 będzie miało w większym stopniu wpływ na efektywność całej linii niż miałoby to w przypadku awarii, przestoju lub spowolnienia maszyn B2 lub B3. Dla linii produkcyjnej C wpływ na efektywność całej linii uzależniona jest od szybkości pracy jej operatorów. Jeśli awarii ulegnie maszyna C1 i operator musi się zająć jej naprawą, to chwilowy przestój zanotuje tylko maszyna C2 obsługiwana przez tego samego operatora, a więc efektywność całej linii obniży się, ale nie w takim stopniu jakby to miało miejsce dla linii produkcyjnej A.

1.1. Cel pracy

Głównym celem pracy jest opracowanie koncepcji analizy efektywności linii produkcyjnych z uwzględnieniem jej specyficznej budowy. Zostaną wykorzystane technologie informatyczne pozwalające na implementacje bibliotek definiujących specyfikacje dla konkretnej linii produkcyjnej.

Pośrednim celem jest opracowanie prototypu aplikacji, która będzie prezentowała przykładowe rozwiązania, jak również charakteryzowała się generycznością poprzez możliwość dodawania nowych pluginów.

1.2. Rozwiązanie przyjęte w pracy

Prototyp systemu informatycznego jest aplikacją desktopową napisaną w języku Java. Implementacja odbyła się przy użyciu środowiska programistycznego Eclipse JUNO. Dane podlegające analizie gromadzone są w bazie danych mySQL. Korzystając z biblioteki Hibernate zapewniono odpowiednią translację obiektowo-relacyjną pomiędzy stworzonym systemem, a bazą danych. Używając biblioteki JSPF umożliwiono zaimplementowanie w systemie narzędzi pozwalających na współpracę z dostarczanymi wtyczkami. W celu zaprezentowania danych końcowych prowadzonej analizy, wykorzystano zewnętrzną bibliotekę JFreeChart umożliwiającą generowanie wykresów graficznych.

1.3. Rezultaty pracy

Bezpośrednim rezultatem pracy będzie przedstawienie zasad analizy współczynnika efektywności w odniesieniu do określonej linii produkcyjnej. Przedstawione zostaną funkcjonalności jakie powinny cechować system umożliwiający przeprowadzenie takiej analizy. Przedstawione zostaną również narzędzia i technologie umożliwiające stworzenie takiego systemu.

Pośrednim rezultatem pracy będzie stworzenie prototypu systemu, który realizując założoną funkcjonalność, pozwoliłby na dokonywanie określonych analiz przy wykorzystaniu pluginów.

1.4. Organizacja pracy

W rozdziale drugim opisywanej pracy przedstawiono niektóre z istniejących rozwiązań systemów umożliwiających badanie efektywności urządzeń i linii produkcyjnych. Dokonano ich scharakteryzowania oraz opisano najważniejsze funkcjonalności analizując ich wady oraz zalety.

Rozdział trzeci zawiera proponowany przez autora projekt generycznego systemu. Przedstawiono w nim funkcjonalności, w jakie należy wyposażyć tworzony prototyp tak, aby umożliwiał on odpowiednie gromadzenie danych produkcyjnych. Ponadto zaprezentowano, wymagany w implementacji nowych pluginów, algorytm pozwalający na przeprowadzenie analizy efektywności linii produkcyjnej.

W rozdziale czwartym dokonano charakterystyki narzędzi oraz technologii wykorzystanych podczas implementacji proponowanego systemu.

Rozdział piąty poświęcony jest na opisanie sposobów implementacji najistotniejszych elementów stworzonego systemu. Zaprezentowano narzędzia pozwalające na wykonanie mapowania obiektowo-relacyjnego, zarządzanie pluginami oraz sposób prezentacji danych końcowych przeprowadzonej analizy.

Rozdział szósty stanowi podsumowanie pracy polegające na opisaniu wad i zalet przygotowanego prototypu systemu oraz przedstawienie możliwych przyszłych kierunków jego rozwoju.

2. Przegląd istniejących rozwiązań

Istnieje wiele rozwiązań systemów informatycznych mających na celu monitorowanie i analizowanie danych napływających z linii produkcyjnych. W zależności od złożoności procesu produkcyjnego użytkownicy mają możliwość wyboru narzędzi tak, aby w jak najlepszy sposób spełniały one wymagania zarówno pod kątem analityczno-biznesowym jak również samego procesu wytwórczego stosowanego w danej fabryce.

2.1. GOLEM OEE SuperVisor

System o nazwie GOLEM OEE SuperVisor (patrz [2]) jest zaawansowanym produktem zaprojektowanym przez firmę NEURON. Jest to rozwiązanie komercyjne, dostępne tylko po zakupieniu odpowiedniej licencji. Producenci oprogramowania nie udostępniają wersji testowej, a więc poniższa analiza systemu przeprowadzona została na podstawie materiałów źródłowych znajdujących się na stronie internetowej producenta. Opisywany system umożliwia pobieranie danych do integracji systemu na dwa sposoby:

- system i urządzenie produkcyjne połączone są ze sobą za pomocą sterowników PLC tak, że system samodzielnie śledzi pracę, przestoje i oblicza efektywność produkcyjną dla tego urządzenia,
- dane potrzebne do analizy trafiają do systemu dostarczane przez operatorów, a cały system nie jest zintegrowany bezpośrednio z urządzeniem.

Do najważniejszych zadań systemu należy:

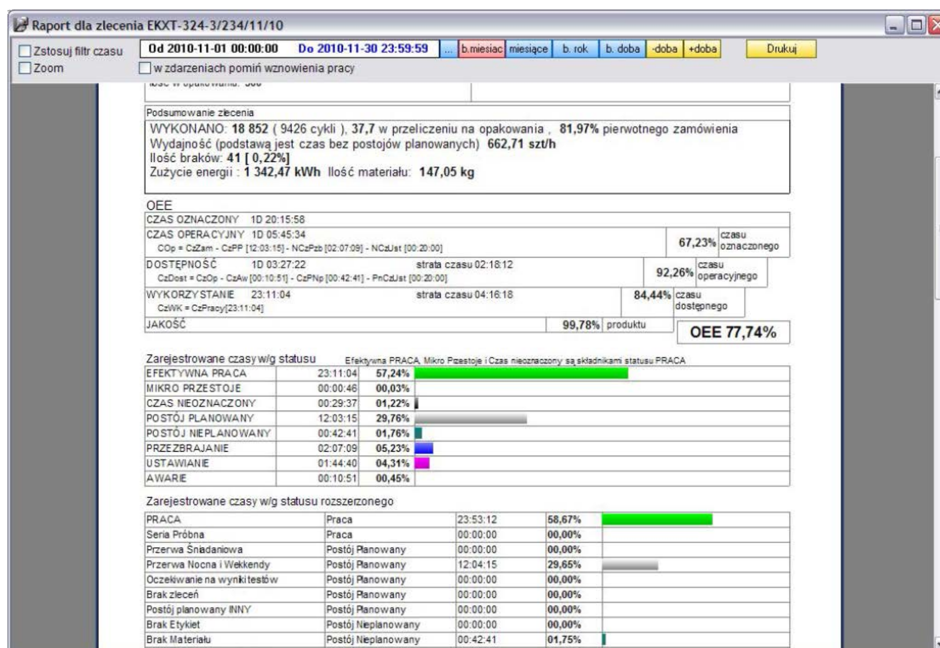
- monitorowanie stanu maszyny wraz z rejestracją wszystkich czasów przestojowych i przyczyn przestojów,
- gromadzenie i analizowanie danych dotyczących ilości wyprodukowanego towaru,
- prezentowanie szczegółowych raportów dotyczących aktualnie wykonywanych zleceń produkcyjnych,
- gromadzenie danych dotyczących operatorów obsługujących daną maszynę w określonym czasie,
- obliczanie i prezentowanie w czasie rzeczywistym parametrów wskaźnika OEE dla określonego urządzenia[1],
- szczegółowe raporty i graficzna prezentacja dotycząca danych potrzebnych do prawidłowego zmierzenia efektywności urządzenia.

Dzięki tym cechom system GOLEM OEE *SuperVisor* pozwala w sposób bardzo dokładny na kontrolowanie i prezentowanie efektywności urządzeń biorących udział w procesie produkcyjnym, jak również zestawianie ogólnych raportów.

Przykładową prezentację wyników analizy przedstawiono poniżej, na rysunku 3 oraz rysunku 4.

[1] Obliczanie wskaźnika OEE w czasie rzeczywistym jest możliwa tylko w przypadku, kiedy system i urządzenie są ze sobą bezpośrednio połączone.

Rysunek 3 Graficzna prezentacja wskaźnika OEE dla określonego urządzenia – GOLEM OEE SuperVisor. Źródło: www.neuron.com.pl.



Rysunek 4 Raport efektywności dla określonego zlecenia - GOLEM OEE SuperVisor. Źródło: www.neuron.com.pl.

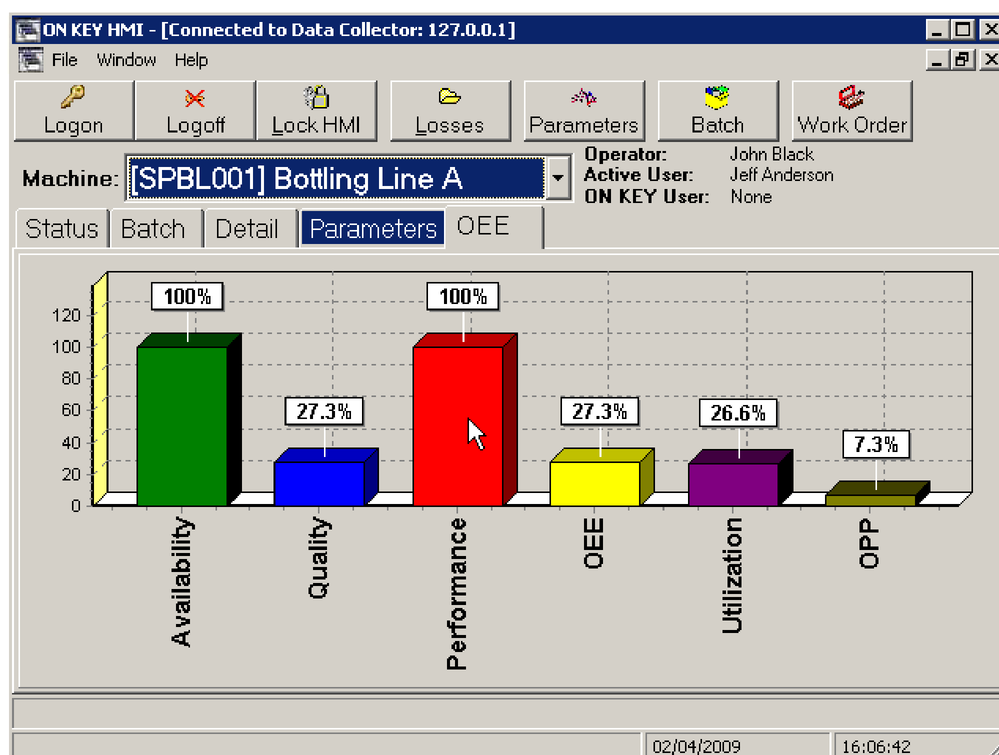
Możliwość gromadzenia zróżnicowanych danych sprawia, że jest on bardzo dobrym narzędziem do analizy efektywności urządzeń wchodzących w skład systemu produkcyjnego. Autorzy systemu zapewniają, że spełnia on swoje zadania zarówno dla dużych i małych

przedsiębiorstw, jednak nie określają dokładnie możliwości, jakie daje system przy analizie całych linii produkcyjnych. W materiałach udostępnionych przez producenta skupiono się jedynie na zobrazowaniu działania systemu skonfigurowanego dla określonego urządzenia produkcyjnego, tak wiec niejasna jest metoda badań efektywności przy zastosowaniu go do specyficznej linii produkcyjnej.

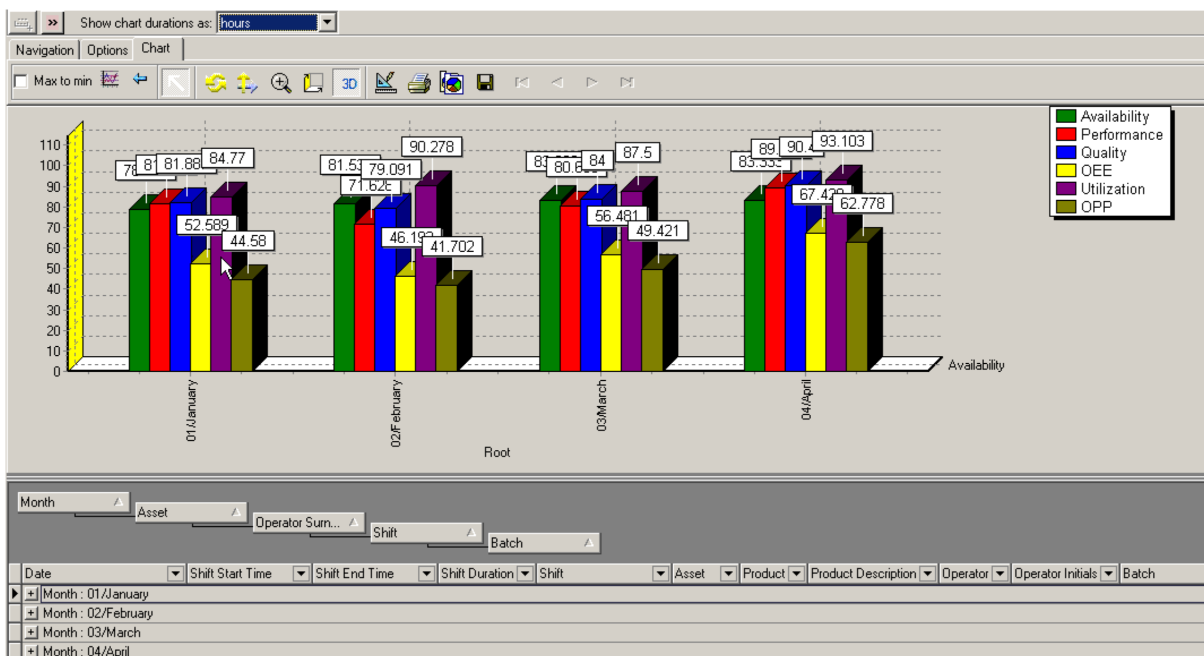
2.2. Computerised Maintenance Management System firmy ESS Ltd.

Computerised Maintenance Management System (patrz [3]) firmy ESS jest kolejnym przykładem zastosowania systemu informatycznego w monitorowaniu efektywności linii produkcyjnych. Jest on również rozwiązaniem komercyjnym dostępnym tylko dla użytkowników posiadających zakupiona licencję. Autorzy systemu na swojej stronie internetowej nie udostępniają wersji demonstracyjnej, a jedynie prezentację multimedialną, w której opisują jego działanie.

System ten zaopatrzony został w dobrze rozbudowany moduł pozwalający na generowanie raportów i śledzenie w czasie rzeczywistym statystyk określających efektywność linii produkcyjnej, co zaprezentowane zostało na rysunku 5. Opierając się na danych historycznych umożliwia on również podobną analizę w odniesieniu do określonego przedziału czasu, w sposób przedstawiony na rysunku 6. Dane poddawane analizie napływają bezpośrednio z urządzenia stanowiącego część całości linii lub bazy danych.

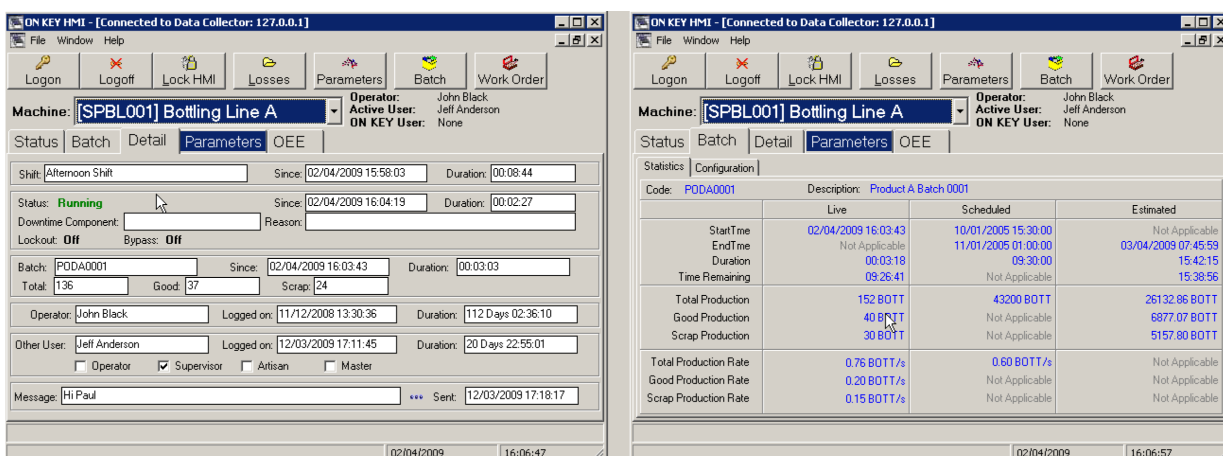


Rysunek 5 Wykresy wskaźnika OEE na podstawie danych napływających w czasie rzeczywistym – CMMS firmy ESS Ltd. Źródło: www.neuron.com.pl.



Rysunek 6 Wykresy wskaźnika OEE na podstawie danych historycznych – CMMS firmy ESS Ltd. Źródło: www.neuron.com.pl.

Pracując w trybie online aplikacja ta dostarcza również szczegółowych informacji na temat aktualnie pracujących na linii operatorów, czasu rozpoczęcia i zakończenia przez nich pracy, bieżącego zlecenia, informacji produkcyjnych takich jak planowanej, aktualnej ilości wyprodukowanego materiału z uwzględnieniem ilości wadliwych produktów. Przykład działania opisywanej powyżej funkcjonalności zaprezentowano na rysunku 7.



Rysunek 7 Dane szczegółowe linii produkcyjnej – CMMS firmy ESS Ltd. Źródło: www.neuron.com.pl.

Oprócz zaprezentowanych powyżej możliwości, system ten wyposażono w szereg typowych dla systemów CMMS funkcjonalności takich jak:

- zarządzanie aktywami danej firmy z możliwością grupowania w taki sposób aby odzwierciedlały one rzeczywistą hierarchię np.: rodzaj działalności → budynki wchodzące w skład danej działalności → wydziały → linie produkcyjne → urządzenia na liniach produkcyjnych,

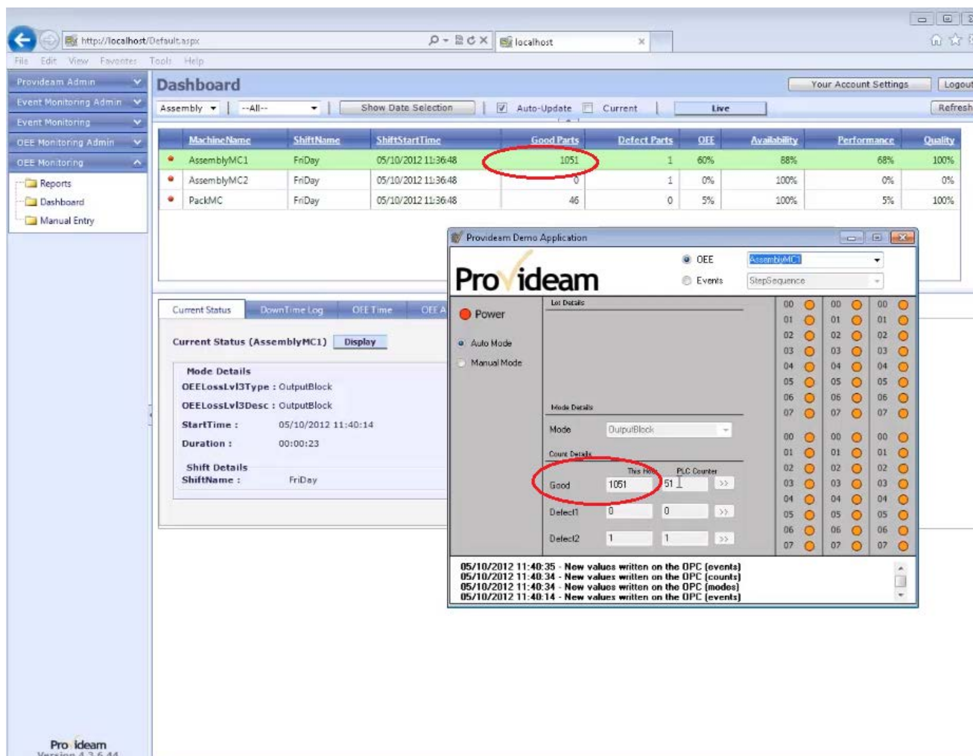
- zarządzanie raportami dotyczącymi awarii, jakie powstały w danym punkcie produkcyjnym dla określonego urządzenia,
- zarządzanie zasobami części zamiennych potrzebnych do naprawy sprzętu,
- tworzenie list pracowników z możliwością grupowania ich pod względem określonych kwalifikacji, departamentu itd.,
- analizę finansową budżetu firmy w postaci raportów i wykresów.

Na podstawie materiałów udostępnionych przez producenta, nie można w sposób jednoznaczny określić, czy istnieje możliwość wdrożenia tego systemu bez konieczności bezpośredniej integracji go z urządzeniami wchodzącymi w skład linii produkcyjnej, tak aby dane dostarczane do analizy pochodziły ze zgłoszeń tworzonych i przesyłanych np.: przez operatorów obsługujących linii. Jeżeli dla omawianego systemu funkcjonalność taka nie jest przewidziana to wdrożenie go wiązałoby się z generowaniem dodatkowych kosztów związanych z zakupem i instalacją dodatkowego oprzyrządowania umożliwiającego integrację systemu z linią produkcyjną.

2.3. Provideam OEE

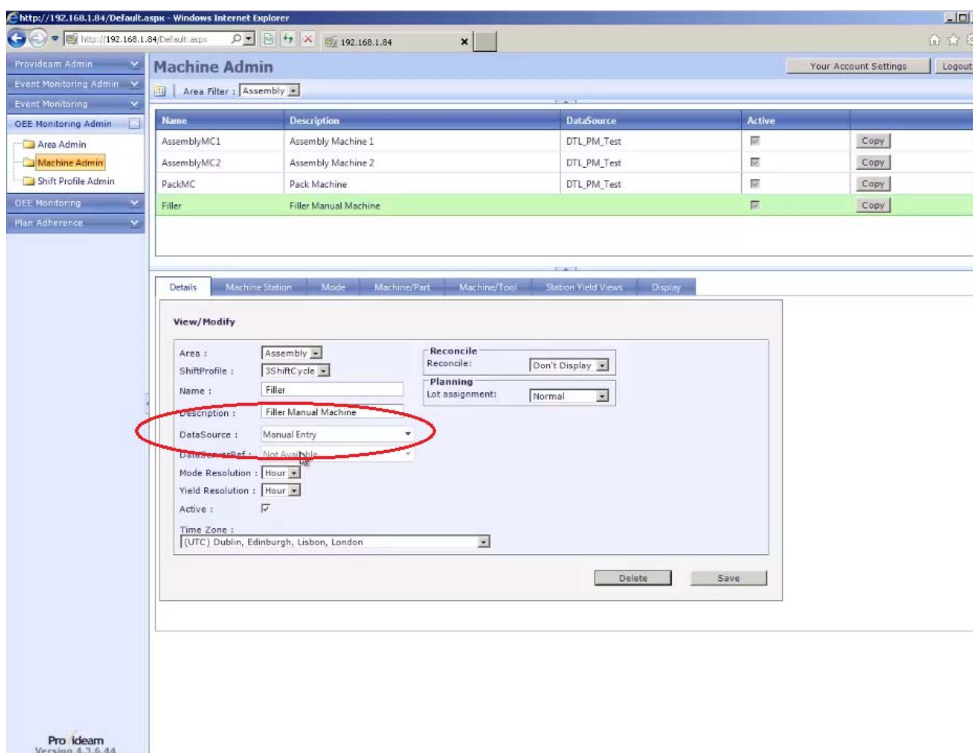
Provideam OEE (patrz [4]) firmy DTL Systems Ltd. jest rozwiązaniem, którego głównym celem jest monitorowanie przestojów na liniach produkcyjnych. Producent na swojej stronie internetowej udostępnia obszerne prezentacje multimedialne dotyczące instalacji i obsługi swojego systemu, jak również wersje demonstracyjna wraz z aplikacja pomocniczą (Provideam Demo Application) symulującą pracę urządzenia produkcyjnego w czasie rzeczywistym (rysunek 8).

Bezpośrednie zintegrowanie systemu i urządzenia produkcyjnego możliwe jest przy pomocy sterowników PLC. Pozwala to, na śledzenie zmian zachodzących na linii produkcyjnej w czasie rzeczywistym dla każdego urządzenia biorącego udział w procesie produkcyjnym. Bezpośrednio napływające dane dają możliwość stałej kontroli wytwarzania, a utrwalanie ich w bazie danych systemu pozwala na późniejszą analizę w szerszym oknie czasowym.



Rysunek 8 Symulacja pracy system w czasie rzeczywistym – Provideam OEE.

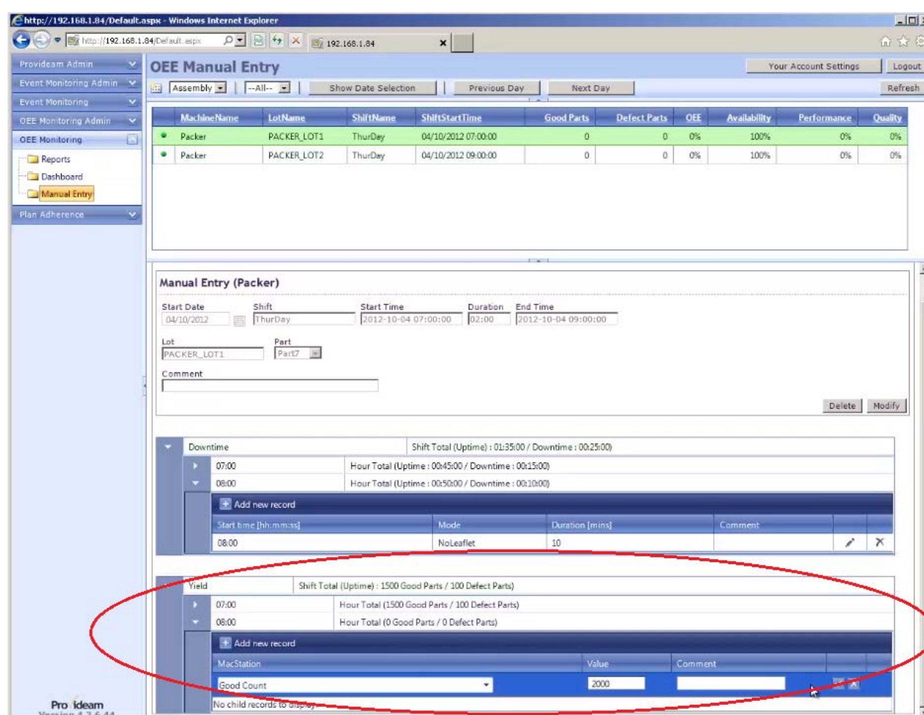
Źródło: www.provideam.com.



Rysunek 9 Konfigurowanie źródła danych dla określonego urządzenia – Provideam OEE.
Źródło: www.provideam.com.

Istnieje również możliwość skonfigurowania systemu tak, aby urządzenia nie musiały być bezpośrednio integrowane z systemem, a dane potrzebne do analizy efektywności można było wprowadzać ręcznie korzystając z dostarczonego przez producenta interfejsu (rysunek 9).

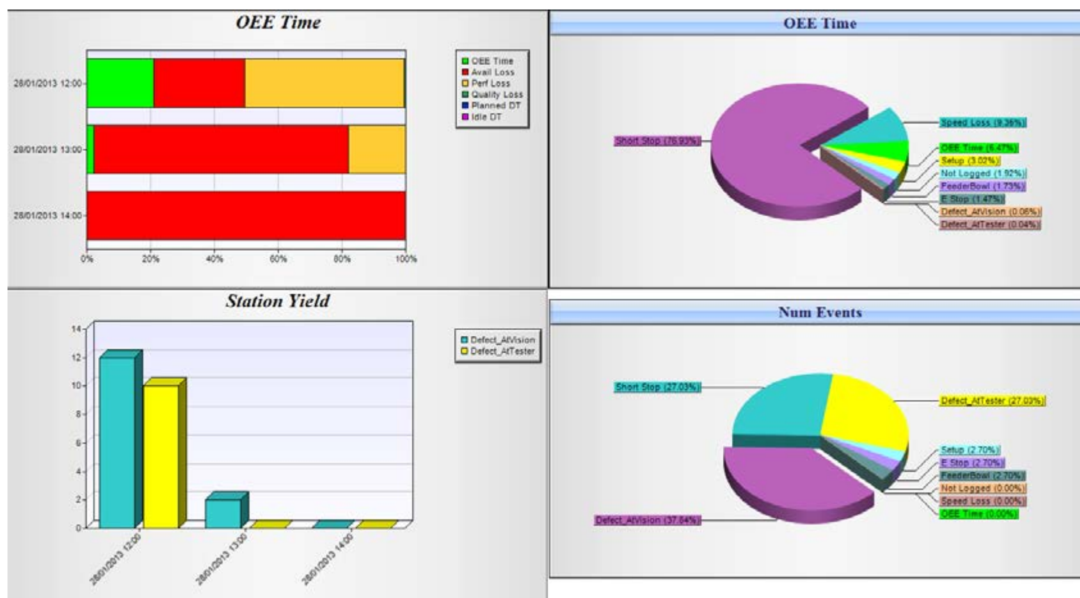
Na rysunku 10 zaprezentowano interfejs przy pomocy którego użytkownik ma do wyboru dwie sekcje wprowadzania danych. Pierwsza dotyczy zgłaszania czasów przestojów, w którym należy określić dokładna godzinę zdarzenia, odpowiedni określający rodzaj przestoju, długość jego trwania i opcjonalnie - komentarz do zdarzenia. Druga sekcja służy do wpisywania ilości wyprodukowanych przez urządzenie dobrych i wadliwych części.



Rysunek 10 Interfejs użytkownika umożliwiający dodawania danych produkcyjnych – Provideam OEE. Źródło: www.provideam.com.

System zapewnia również możliwość tworzenia szeregu wykresów i raportów w zależności od urządzenia produkcyjnego, grupy urządzeń, areny w jakiej to urządzenie się znajduje, danych jakich potrzebuje w określonych przez siebie przedziale czasowym. Umożliwia to dostarczony przez producenta interfejs za pomocą, którego użytkownik sam określa parametry jakie chce analizować. Rezultat tworzonych wykresów i raportów zaprezentowano na rysunku 11.

Z uwagi na swoją komercyjność produkt ten jest stworzony tak, aby zapewnić swoją funkcjonalność dla najszerszego grona odbiorców. Dostęp do interfejsu użytkownika odbywa się za pośrednictwem przeglądarki internetowej. Posiada on szereg funkcji dających możliwość dostosowania interfejsu do profilu firmy oraz odpowiedniej grupy użytkowników. Niedoskonałością systemu jest, iż mimo możliwości tworzenia szeregu różnych rodzajów raportów mogą one dotyczyć tylko określonego urządzenia lub urządzeń należących do określonej grupy nie dając tym samym opcji analizowania danych z perspektywy całej linii produkcyjnej.



Rysunek 11 Wykresy wskaźnika OEE oraz produkcji dla wybranego urządzenia – Provideam OEE. Źródło: www.provideam.com.

2.4. System SMLP

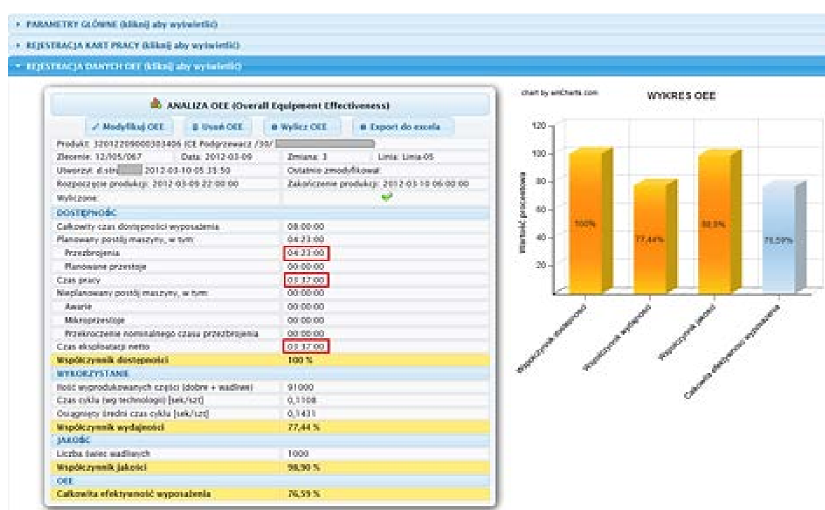
System SMPL (System Monitorowania Linii Produkcyjnej) (patrz [5]) stworzony przez firmę Progresja Konsulting, jest narzędziem dającym możliwość rejestracji i analizy zdarzeń zachodzących na liniach produkcyjnych. Dostęp do aplikacji i serwera danych odbywa się za pośrednictwem przeglądarki internetowej.

Zbudowany modułowo system, którego główny panel użytkownika ukazano na rysunku 12, odpowiada między innymi za:

- rejestrację mikro-przestojów, awarii oraz przezbrojeń,
- rejestrację zleceń zgłaszanych do działu utrzymania ruchu,
- rejestrację planowanych przestojów,
- raportowanie i tworzenie wykresów dotyczących zebranych danych.

Rysunek 12 Główny panel użytkowania – SMLP. Źródło: www.progresja.com.pl.

Ponadto system pozwala na wprowadzenie danych produkcyjnych (na przykład ilość dobrych i uszkodzonych produktów), co umożliwiłoby późniejszą analizę efektywności danej linii w stosunku do danego zlecenia produkcyjnego, daty czy zmiany. Zaprezentowany poniżej rysunek 13 przedstawia przykładowy raport wykonany przez system SMLP.



Rysunek 13 Raport dotyczący wskaźnika OEE – SMLP. Źródło: www.progresja.com.pl

Twórcy systemu SMPL, poprzez nieudostępnienie wersji testowej, uniemożliwili przeprowadzenie wnikliwej analizy oferowanego produktu[2]. Istnieje wiele niejasności związanych z kalkulacją wskaźnika OEE, określaniem czasów przestoju oraz z przygotowaniem samego systemu pod określoną linię produkcyjną.

2.5. Podsumowanie

W rozdziale tym przedstawiono kilka systemów, których celem jest analiza płynności produkcyjnej. Pomimo dużej liczby istniejących rozwiązań autor tej pracy nie znalazł systemu,

[2] Wszystkie informacje zawarte w pracy magisterskiej nt. systemu SMLP pochodzą ze strony internetowej producenta

który pozwoliłby na wyznaczenie wskaźnika efektywności w perspektywy całej linii produkcyjnej. Większość rozwiązań skupia się na szacowaniu tego parametru wyłącznie pod kątem określonych urządzeń, tak więc wyznaczenie współczynnika OEE całej linii produkcyjnej związane byłoby z przeprowadzeniem dodatkowych obliczeń. W przypadku standardowej budowy linii, jak to zostało zaprezentowane na rysunku 2 w przykładzie A, współczynnik OEE jest równy najniższemu współczynnikowi OEE urządzenia tworzącego tą linię. Natomiast w przypadku bardziej złożonej konstrukcji linii, analiza jej efektywności nie jest jednoznaczna więc musi być rozpatrywana indywidualnie.

Prototyp zaproponowanego przez autora systemu pozwoliłby nie tylko na obliczanie współczynnika efektywności dla danego urządzeń, ale również dla zestawu urządzeń pracujących jako jedna linia produkcyjna, przy uwzględnieniu jej specyficznej budowy. Z biznesowego punktu widzenia zapewniłby również dokładniejszą analizę efektywności produkcji związaną z konkretnym zleceniem produkcyjnym, produktem lub zmianą.

3. Założenia proponowanego rozwiązania

Projekt zakłada stworzenie generycznego systemu badającego efektywność linii produkcyjnych przy użyciu zewnętrznych wtyczek (pluginów). Przedstawione w rozdziale 2 istniejące rozwiązania, opierają się na wykorzystaniu sterowników PLC. Służą one do monitorowania całej linii produkcyjnej, określonych urządzeń, które wchodzą w jej skład i do wyliczania współczynnika OEE.

Proponowane rozwiązanie, przedstawione w poniższym rozdziale, ukazuje system analizujący współczynnik efektywności linii produkcyjnej bez konieczności wykorzystania dodatkowych urządzeń monitorujących.

3.1. Program główny

Rodzaj danych umożliwiający przeprowadzenie analizy wyżej opisywanego współczynnika jest taki sam dla każdego typu linii produkcyjnej. Do wielkości reprezentujących takie dane należą:

- **całkowity czas produkcji** – jednostka czasu odnośnie której dokonywana jest analiza efektywności. Przykładem takiego okna może być np.: zmiana, dzień, miesiąc,
- **prędkość maksymalna** (*Target*) – wielkość określająca maksymalną ilość produktów możliwych do wytworzenia przez dane urządzenie w określonym oknie produkcyjnym,
- **przestój** (*Downtime*) – całkowity czas przestoju danego urządzenia w analizowanym oknie produkcyjnym,
- **produkty wadliwe** (*Rejects*) – ilość wadliwie wytworzonych produktów przez określoną maszynę w analizowanym oknie produkcyjnym,
- **wyprodukowana ilość** (*Output*) – liczba prawidłowych produktów wytworzonych przez dane urządzenie w analizowanym oknie produkcyjnym.

Opisane powyżej dane powinny odnosić się do poszczególnych urządzeń wchodzących w skład linii produkcyjnej. Dzięki temu w dokładny sposób możliwe jest odwzorowanie procesów zachodzących w każdym urządzeniu.

Opierając się na takich założeniach, projektowany system powinien posiadać funkcjonalności umożliwiające:

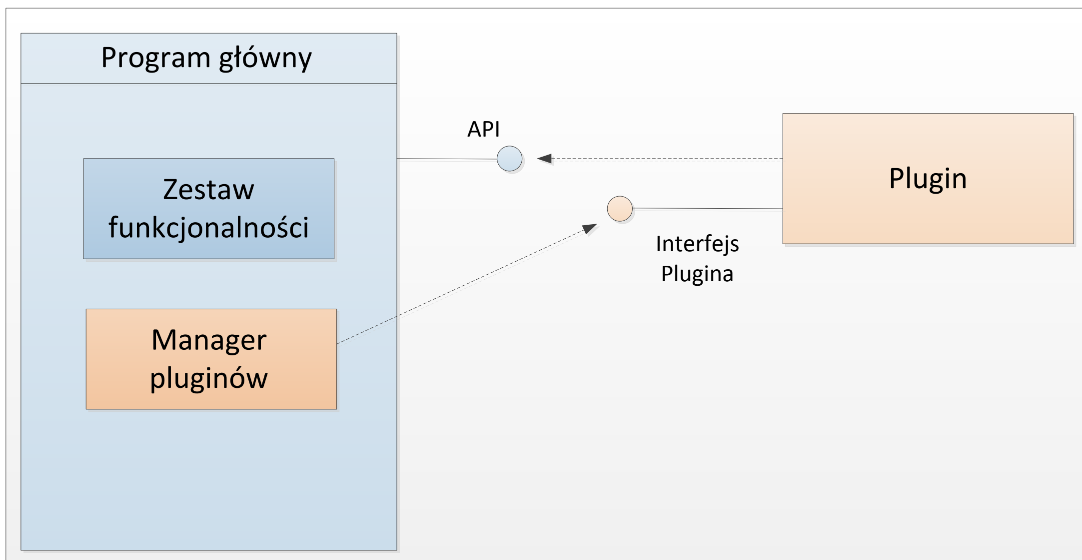
- definiowanie linii produkcyjnych,
- definiowanie urządzeń, będących elementami linii produkcyjnej,
- przypisywanie określonych urządzeń do określonej linii produkcyjnej,
- definiowanie produktów wytwarzanych na poszczególnych urządzeniach,
- przypisywanie produktów do określonych urządzeń,
- usuwanie oraz edytowanie istniejących produktów, urządzeń oraz linii produkcyjnych,
- zarządzanie użytkownikami systemu,
- tworzenie raportów dotyczących określonego urządzenia zawierające wyżej wymieniony zestaw danych.

Stworzenie systemu wyposażonego w wyżej wymienione funkcje, pozwoliłby na wykorzystanie go do analizy każdego rodzaju linii produkcyjnej, abstrahując od profilu firmy, czy rodzaju wytwarzanego przez nią produktu.

3.2. Modułowość projektowanego system

Opisane w punkcie 3.1 dane produkcyjne są niestety niewystarczające do przeprowadzenia analizy współczynnika OEE w ujęciu całej linii produkcyjnej. Pozwalają one jedynie na przeprowadzenie analizy na poziomie poszczególnych urządzeń. Odnosząc się do całości linii istotnym czynnikiem wpływającym na wiarygodność współczynnika efektywności jest również rozkład maszyn oraz relacje jakie pomiędzy nimi zachodzą. Wymagane jest zatem, aby prototyp tworzonego systemu posiadał możliwość dostarczenia do niego odpowiedniej definicji takich zależności.

Proponowanym rozwiązaniem realizującym ową funkcjonalność jest zaimplementowanie w tworzonym prototypie systemu odpowiedniego API (*Application Programming Interface*), który umożliwiłby współpracę z dostarczanymi z zewnątrz pluginami. Tworzone, w odniesieniu do konkretnej linii produkcyjnej pluginy, zawierałyby odpowiednią definicję rozkładu urządzeń oraz zachodzących pomiędzy nimi relacji. Schemat ukazujący realizację opisywanego rozwiązania przedstawiono na rysunku 14.



Rysunek 14 Schemat współpracy systemu głównego z pluginami. Źródło: Opracowanie własne

3.3. Analiza danych

Opisywany we wstępie współczynnik OEE wyliczany jest za pomocą parametrów opisanych jako „dostępność", „wydajność" oraz „jakość". Każdy z nich posiada własny niezmienny algorytm obliczeniowy oparty na gromadzonych danych produkcyjnych uwzględniających relacje zachodzące pomiędzy poszczególnymi elementami linii produkcyjnej. W celu przeprowadzenia odpowiedniej analizy efektywności, każdy z utworzonych pluginów wyposażyć należy w metody umożliwiające ich wyliczanie, w odniesieniu do całej linii produkcyjnej. Poniżej zdefiniowane zostały wzory umożliwiające wyznaczanie poszczególnych parametrów współczynnika OEE.

„Dostępność" to wielkość uwzględniająca wszystkie planowane i nieplanowane przestoje linii produkcyjnej w stosunku do całkowitego czasu produkcji. Wyrażana jest ona następującym wzorem:

$$A = T_o \div T_{pl}$$

, gdzie:
T_o – wielkość określana jako czas operacyjny,
T_{pl} – wielkość określana jako planowany czas produkcji.

Planowany czas produkcji T_{pl} definiowany jest jako:

$$T_{pl} = T_t - T_{pp}$$

, gdzie:
T_t – całkowity czas produkcji. Parametr opisywany na początku tego rozdziału (patrz 3.1) wyrażający jednostkę czasu, według której dokonywana jest analiza efektywności,
T_{pp} – całkowity czas przeznaczony na przestoje planowane. Można do nich zaliczyć np.: przerwy śniadaniowe lub planowany serwis, jednak dokładna interpretacja tej wielkości powinna znajdować się w pluginie zaprojektowanym dla określonej linii produkcyjnej.

Drugim parametrem pozwalającm na określenie dostępności jest czas operacyjny T_o wyrażony wzorem:

$$T_o = T_{pl} - T_{np}$$

, gdzie:
T_{pl} – planowany czas produkcji,
T_{np} – całkowity czas nieplanowanego przestoju linii produkcyjnej w przeznaczonym czasie produkcyjnym (T_t). Wielkość ta powinna być również dokładnie zdefiniowania w implementowanym pluginie, ponieważ nieplanowany czas przestoju niektórych maszyn nie musi zawsze oznaczać przestoju całej linii, a jedynie wyłączenie z użytku jej części co wiąże się z opisaną poniżej stratą prędkości.

Kolejnym parametrem wymaganym do obliczenia współczynnika OEE jest „wydajność". Pozwala ona na oszacowanie strat produkcyjnych wynikających ze zmniejszenia prędkości pracy danej linii. Wielkość ta wyrażana jest następującym wzorem:

$$P = T_{netto} \div T_o$$

, gdzie:
T_{netto} – czas operacyjny netto,
T_o – czas operacyjny.

Czas operacyjny netto (T_{netto}) jest wielkością uwzględniającą straty prędkości danej linii poprodukcyjnej i wyrażana jest wzorem:

$$T_{netto} = T_o - LR$$

, gdzie:
T_o – czas operacyjny,
LR – strata prędkości jest iloczynem czasu w jakim urządzenie pracowało ze zmniejszoną prędkością do jego wydajności. Przykładowo spowolnienie maszyny do 10% przez okres 40 minut może być traktowane jako całkowite jej zatrzymanie przez 4 minuty. W implementowanym pluginie powinna się zatem znaleźć definicja określająca wpływ przestoju takiej maszyny na wydajność linii produkcyjnej, przy założeniu, że przestój ten jest uwzględniany przy obliczaniu parametru „wydajności", a nie „dostępności".

Ostatnim parametrem wpływającym na współczynnik efektywności jest „jakość". Uwzględnia on straty wynikające z wytwarzania wadliwych produktów i wyrażony jest wzorem:

$$Q = T_{ep} \div T_{netto}$$

,gdzie:
T_{ep} – współczynnik określane jako czas efektywnej produkcji,
T_{netto} – przedstawiony powyżej współczynnik określany jako czas operacyjny netto.

Czas efektywnej produkcji jest wyrażana poprzez:

$$T_{ep} = T_{netto} - L_Q$$

, gdzie:
T_{netto} czas operacyjny netto,
L_Q – straty jakościowe - wielkość określająca ilość wyprodukowanych przez dane urządzenie wadliwych produktów w stosunku do wydajności takiego urządzenia. Zakładając, że urządzenie pracujące z wydajnością 10 sztuk/minutę w ciągu godziny wytworzyło 40 wadliwe sztuki, straty jakościowe w takim przypadku wyniosły 4 minuty. Czas ten, traktowany jako przestój maszyny i uwzględniany jest w implementowanym pluginie przy obliczaniu parametru „jakość”.

Jak już wcześniej wspomniano, w zależności od rozkładu urządzeń, dane produkcyjne w odniesieniu do całej linii traktowane mogą być w różny sposób. Przestój maszyny w jednym przypadku może oznaczać unieruchomienie całej produkcji, a w innym tylko jej spowolnienie. Duże znaczenie ma również to, czy dana maszyna wchodząca w skład linii produkcyjnej jest obsługiwana przez operatora, czy może działa samoczynnie. W takim wypadku przerwy śniadaniowe brane przez obsługę maszyny muszą być uwzględnione jako zaplanowana przerwa w produkcji, ale już przerwa w działaniu urządzenia spowodowana tym, że operator ma spotkanie z przełożonym kwalifikowana powinna być jako niezaplanowany przestój maszyny. Dodatkowo wystąpienie takiej sytuacji nie zawsze może oznaczać zatrzymanie całej linii produkcyjnej, a więc w takim przypadku rozważać należy to w kategoriach spowolnienia produkcji.

Opisane przypadki stanowią jedynie przykłady możliwych relacji, a dokładna ich identyfikacja powinna odbyć się na etapie projektowania danego pluginu. Zawarta w nich implementacja dobrze zdefiniowanych zależności w połączeniu z danymi produkcyjnymi pochodzącymi z określonych urządzeń, pozwoli na właściwe oszacowanie żądanego współczynnika OEE całej linii produkcyjnej. Zadaniem plugina będzie również przekazanie do programu głównego przeanalizowanych danych w formie określonego modelu danych umożliwiając w ten sposób zaprezentowanie wyniku w formie wykresu graficznego.

4. Zastosowane narzędzia i technologie

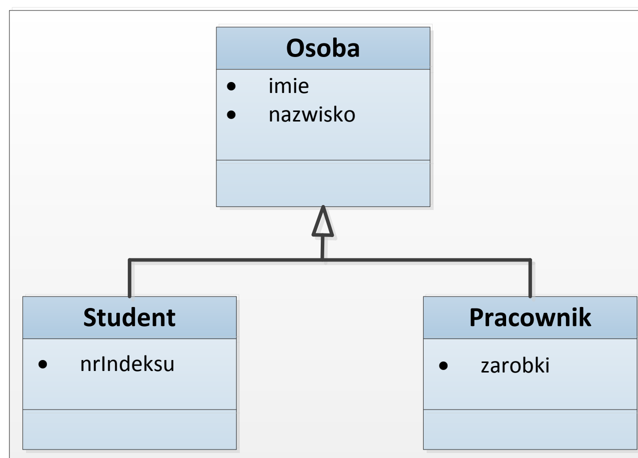
Rozdział ten poświęcono omówieniu narzędzi i technologii, jakie zostały użyte w trakcie tworzenia oraz implementacji generycznego systemu informatycznego analizującego efektywność linii produkcyjnych.

4.1. Język Java

Początki języka Java (patrz [7],[18],[19]) datuje się na rok 1991, kiedy to pod przewodnictwem Jamesa Goslinga i Patricka Nouthona w ramach projektu *Green* grupa inżynierów z firmy *Sun* rozpoczęła pracę nad stworzeniem oprogramowania przeznaczonego dla urządzeń elektronicznych powszechnego użytku. Urządzenia te charakteryzowały się małą ilością posiadanej pamięci oraz dużym zróżnicowaniem, co do instalowanych w nich procesorów, dlatego też napisanie do nich oprogramowania wymagało zastosowania języka, który poradziłby sobie z tymi ograniczeniami. W rezultacie prowadzonych prac na przestrzeni pięciu lat powstał nowy język programowania o nazwie Java (początkowo nazwany *Oak*).

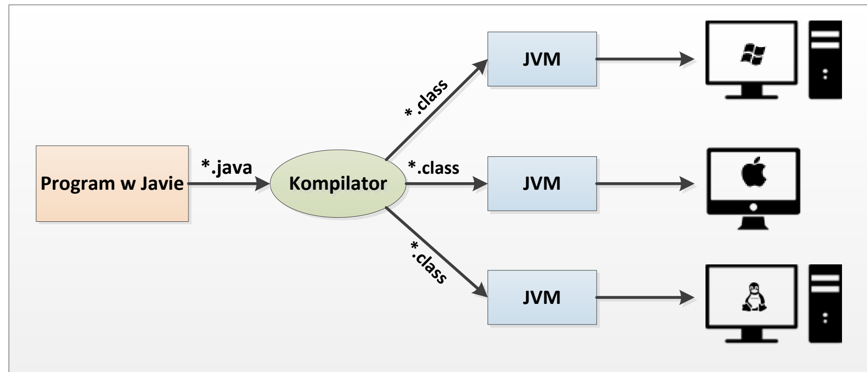
Do najważniejszych cech charakteryzujących język Java należą:

- **obiektowość** – jest jedna z podstawowych i najważniejszych cech języka Java. Oznacza to, że każdy zdefiniowany za pomocą tego języka byt mający odzwierciedlenie w modelowej rzeczywistości reprezentowany jest za pomocą tzw. „Obiektu”. Definiowanie obiektów odbywa się przy użyciu „Klas” zawierających opis pól i metod charakterystycznych dla określonej grupy obiektów. Pola występujące w określonej klasie rozumiane są jako właściwości danego obiektu, natomiast metody jako usługi (czynności) jakie dany obiekt może wykonać,
- **dziedziczenie** – umożliwia tworzenie nowych klas (podklas) w oparciu o inne już istniejące klasy (nadklasy). Dzięki temu nowo tworzona klasa przejmuje właściwości i funkcjonalności klasy, którą dziedziczy oraz umożliwia zdefiniowania nowych, dodatkowych własności. Pozwala to na tworzenie bardziej wyspecjalizowanych obiektów w stosunku do obiektów będących instancjami dziedziczonej klasy. Przykładowy sposób dziedziczenia w postaci diagramu UML zaprezentowano na rysunku 15. Ponadto w języku Java nie jest możliwe wielodziedziczenie, a więc jedna podklasa może dziedziczyć tylko po jednej nadklasie. Jeżeli określona klasa nie ma jasno zdefiniowanej nadklasy oznacza to, że domyślnie dziedziczy ona bezpośrednio po klasie Object, będącej „korzeniem” w hierarchii klas języka Java,



Rysunek 15 Przykład dziedziczenia w Javie. Źródło: Opracowanie własne

- **hermetyzacja** – cecha umożliwiająca ukrycie wybranych własności obiektów tak, aby były one dostępne tylko w obrębie danej klasy. Możliwe jest to za pomocą określonych słów kluczowych nazywanych modyfikatorami dostępu. Oznacza to, że pola lub metody zdefiniowane za pomocą słowa kluczowego „*public*” widoczne są globalnie, a każda instancja może mieć do nich dostęp. Użycie modyfikatora dostępu „*private*” spowoduje, że opisane nim pola lub metody dostępne będą tylko w obrębie danej klasy z wyłączeniem klas dziedziczących. Aby umożliwić dostęp do pól lub metod również klasą dziedziczącym, należy określoną własność zdefiniować za pomocą modyfikatora „*protected*”. Brak użycia jakiegokolwiek modyfikatora dostępu w języku Java domyślnie oznacza, że dostęp do pól lub metod tak zdefiniowanych możliwy jest tylko dla instancji klas należących do tego samego pakietu,
- **polimorfizm** – brak możliwości wielodziedziczenia w języku Java częściowo rozwiązują tzw. interfejsy zawierające abstrakcyjne metody i/lub publiczne statyczne pola zadeklarowane jako *final* . Implementacja interfejsu przez określone klasy wiąże się z koniecznością zaimplementowania w nich metod i pól zadeklarowanych we wspomnianym interfejsie. Sposób implementacji może być odmienny dla każdej z takich klas, a właściwość odpowiadająca za wywołanie właściwej metody na nazywana jest polimorfizmem,
- **przenośność** – stworzenie języka będącego całkowicie niezależnym od architektury na jakiej działa było jednym z podstawowych założeń projektu *Green*. Rozwiązano to za pomocą „wirtualnej maszyny” (*Java Virtual Machine*, JVM), którą należy zainstalować w określonym środowisku, aby zapewnić poprawne działanie programów. Maszyna ta pełni rolę wirtualnego procesora, który wykonuje stworzony podczas kompilacji, kod pośredni zwany *byte-code*, przetwarzając go na instrukcję rozumiane przez określone środowisko programowo-sprzętowe. Dzięki temu programy napisane w języku Java cechują się przenośnością, a raz skompilowane mogą być uruchamiane w każdym środowisku posiadającym odpowiednią wersję wirtualnej maszyny Javy. Na rysunku 16 przedstawiono uproszczony schemat obrazujący opisane powyżej procesy,



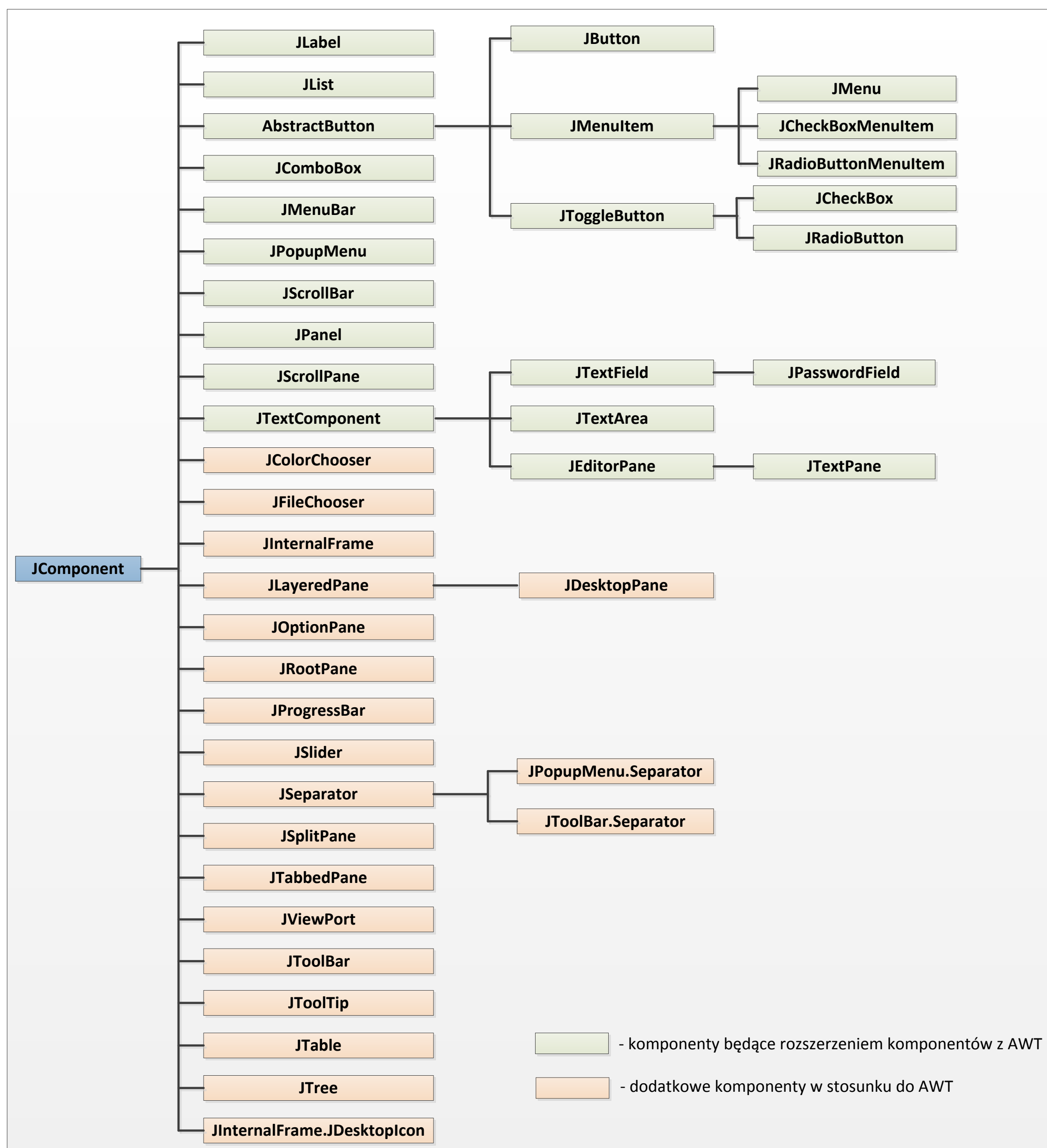
Rysunek 16 Proces uruchamiania programu w języku Java.
Źródło: Opracowanie własne

- **współbieżność** – język Java wspiera również programowanie wielowątkowe. Oznacza to, że w ramach jednego programu możliwe jest równoległe wykonywanie wielu zadań. Takie działanie możliwe jest dla każdej nowo stworzonej klasy, która dziedziczy po klasie o nazwie *Thread* lub implementuje interfejs *Runnable*. Ponadto możliwość synchronizacji pomiędzy określonymi wątkami gwarantuje, że nie będą one działać równocześnie na tym samym obiekcie, co ma znaczenie przy zachowaniu odpowiedniej spójności,
- **silna kontrola typów** – cech ta powoduje, że w języku tym każda wartość posiada określony typ. W Javie istnieją dwa rodzaje typów: typy obiektowe oraz proste. Typy obiektowe definiowane są poprzez zaimplementowane lub dostarczone w postaci bibliotek klasy. Drugim rodzajem jest 8, wbudowanych w język, typów prostych takich jak: char, boolean, byte, short, int, long, float oraz double,
- **obsługa błędów** – dzięki użyciu specjalnych konstrukcji językowych, Java umożliwia wykrycie i obsługę błędów. Wystąpienie błędu podczas wykonującego się kodu prowadzi do stworzenia obiektu reprezentowanego przez określony wyjątek, a następnie zgłoszenie go w metodzie, w której ten błąd wystąpił.

4.2. Biblioteka Swing

Udostępniona od wersji Java 1.2 biblioteka Swing (patrz [7],[8],[18],[19]) stanowi rozszerzenie istniejącej wcześniej w języku Java biblioteki AWT (*Abstract Window Toolkit*). Posiada ona szereg elementów wykorzystywanych do tworzenia interfejsu graficznego (*Graphical User Interface,* GUI) potrzebnego użytkownikowi do komunikowania się z tworzonym systemem.

Podstawowymi elementami tworzącymi graficzny interfejs użytkownika są komponenty. Każdy z takich komponentów posiada specyficzne właściwości i funkcje zaimplementowane w odpowiadających im klasach. Wszystkie one dziedziczą po klasie *JComponent*, posiadającej wspólne właściwości wszystkich komponentów biblioteki Swing. Występujące w opisywanej bibliotece komponenty z uwzględnienie hierarchii dziedziczenia przedstawiono na rysunku 17.

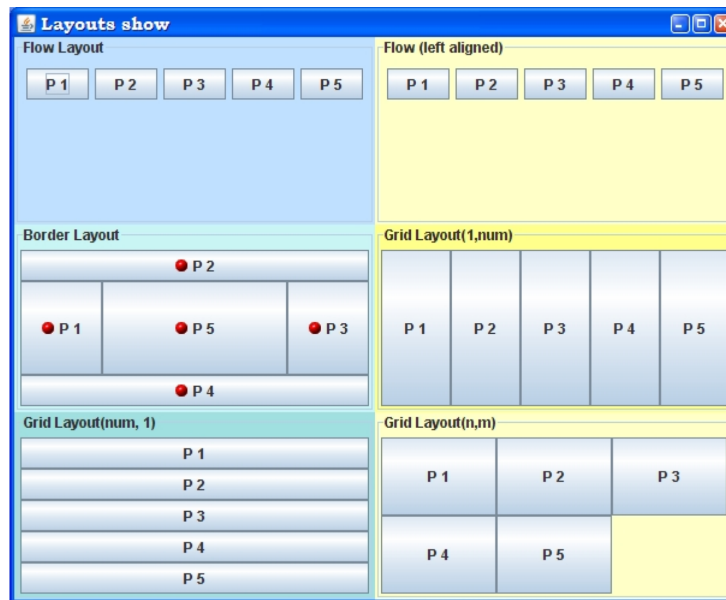


Rysunek 17 Hierarchia komponentów biblioteki SWING. Źródło: Opracowanie własne

Udostępnione przez bibliotekę AWT oraz Swing komponenty, umożliwiające umieszczanie w nich innych komponentów określane są mianem kontenerów. Wywołanie obiektu klasy implementującej interfejs *LayoutManager*, umożliwia stworzenie tzw. zarządcy rozkładu. Pozwala on na odpowiednie rozmieszczenie elementów GUI w obrębie danego kontenera. Korzystając z biblioteki Swing możliwe jest stworzenie następujących rodzajów rozkładu:

- FlowLayout,
- BorderLayout,
- GridLayout,
- BoxLayout,
- CardLayout.

Na rysunku 18 zaprezentowano efekt wykorzystania niektórych z wyżej wymienionych zarządców rozkładu. Ponadto, dla każdego z nich istnieje również szereg metod pozwalających na jeszcze dokładniejsze sprecyzowanie tworzonego przez programistę rozkładu.



Rysunek 18 Przykładowe rodzaje rozkładu komponentów.
Źródło: http://edu.pjwstk.edu.pl/wyklady/poj/scb/index.html.

Interfejs użytkownika oprócz wizualizacji określonych elementów graficznych powinien zapewniać odpowiednią jego interakcję z użytkownikiem. Realizowane jest to za pomocą metod, umożliwiających dodanie do określonego komponentu tzw. słuchacza. Słuchacz jest obiektem implementującym odpowiedni interfejs nasłuchu (*Listener Interface*) posiadający zadeklarowane metody, w których należy zawrzeć kod umożliwiający wykonanie odpowiedniego zdarzenia.

4.3. MigLayout

MigLayout (patrz [9]) jest darmową biblioteką opartą na licencji BSD lub GPL. Umożliwia ona skorzystanie z innego, niż opisywane w punkcie 3.4.1, zarządcy rozkładu. Za jego pomocą, dla określonego kontenera, tworzona jest „siatka" rozkładu składająca się pojedynczych „komórek" do których trafiają poszczególne komponenty GUI.

Szereg dostępnych ograniczeń, formowanych za pomocą wyrażeń typu *String*, pozwala na znacznie bardziej precyzyjne określenie kształtu oraz zachowania się siatki lub jej poszczególnych komórek. Ograniczenia te zadawane mogą być na poziomie kolumn, wierszy, komponentów lub samego rozkładu. Przykładowy fragment kodu tworzący siatkę rozkładu przedstawiono na listingu 1.

Listing 1 Przykład implementacji MigLayout w odniesieniu do wybranego komponentu Swing.

```
JFrame frame = new JFrame("New Frame");
frame.setLayout(new MigLayout("","20[]20[right][grow]","20[][][]20[]"));
```

Ilość wierszy i kolumn określana jest za pomocą wyrażenia złożonego z nawiasów kwadratowych ([]) reprezentując tym samym poszczególne komórki na siatce rozkładu. W zaprezentowanym powyżej przypadku siatka rozkładu będzie posiadała 3 kolumny oraz 4 wiersze. Zastosowanie ograniczenia *right* oznacza, że komponenty znajdujące się we wszystkich komórkach drugiej kolumny będą wyrównywane do prawej krawędzi komórki. Wyrażenie *grow,* pozwoli na rozciąganie i ściskanie tylko tych komórek siatki rozkładu, które będą należały do kolumny trzeciej. Dodatkowo posługując się wartością liczbową (domyślnie wartość wyrażona jest w pikselach) określona została szerokość odstępu pomiędzy niektórymi kolumnami i wierszami. Tak zdefiniowany rozkład przyjmuje postać siatki zaprezentowanej na rysunku 19.

komponent 1	komponent 2	komponent 3
komponent 4	komponent 5	komponent 6
komponent 7	komponent 8	komponent 9
komponent 10	komponent 11	komponent 12

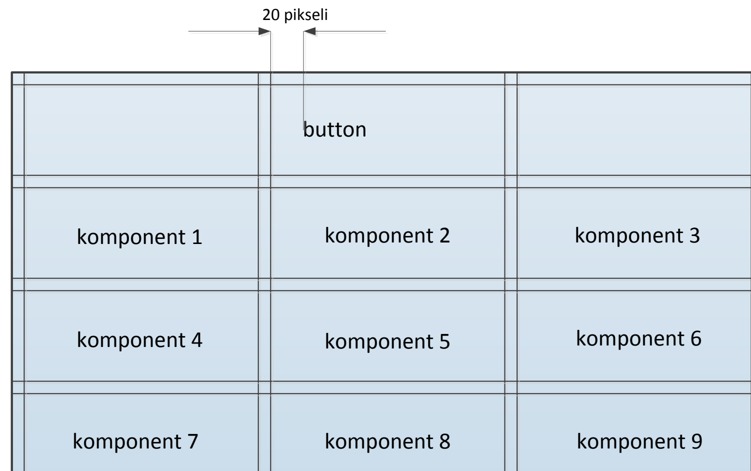
Rysunek 19 Przykładowa siatka rozkładu - biblioteka MigLayout. Źródło: Opracowanie własne

Komponenty GUI w chwili dodawania ich do kontenera umiejscawiane są domyślnie w kolejnych komórkach stworzonej siatki rozkładu. Definiując odpowiednie ograniczenia w postaci dodatkowego argumentu w metodzie add, biblioteka MigLayout umożliwia jeszcze dokładniejsze rozmieszczenie elementów GUI tym razem już na poziomie pojedynczej komórki. Na zaprezentowany poniżej listingu 2 przedstawiono przykładowe zastosowanie ograniczeń dla wybranego elementu GUI.

Listing 2 Przykładowe zastosowania ograniczeń biblioteki MigLayout.

```
JButton button = new JButton("New button");
frame.add(button, "skip 1, gapleft 20, wrap");
```

Za pomocą ograniczenia *skip 1*, komponent umiejscowiony zostanie dopiero w drugiej komórce siatki rozkładu. Wyrażenie *gapleft 20* określa odległość (wyrażona w pikselach) dodanego elementu od lewej krawędzi komórki, a wyrażenie *wrap* oznacza przejście do kolejnego wiersza siatki rozkładu. Oczekiwany efekt tak zdefiniowanego rozkładu zaprezentowano na rysunku 20.



Rysunek 20 Przykładowe umiejscowienie komponentu w siatce rozkładu – MigLayout. Źródło: Opracowanie własne

Dzięki zastosowaniu MigLayout projektowanie graficznego interfejsu w znacznym stopniu poprawia czytelność kodu, a duża ilość dostępnych ograniczeń pozwala na bardzo precyzyjny rozkład komponentów poszczególnych komórek wchodzących w skład siatki rozkładu.

4.4. Java Simple Plugin Framework

Java Simple Plugin Framework (JSPF) (patrz [13]) jest opartą na licencji BDS, darmową biblioteką, pozwalającą na dostosowanie określonego systemu informatycznego do obsługi wtyczek (*plugins*). Biblioteka ta charakteryzuje się małą objętością, dużą szybkością działania oraz prostą implementacją, co z powodzeniem pozwala na stosowanie jej w małej i średniej wielkości projektach.

Szczegóły implementacji biblioteki zostały całkowicie ukryte, a praca z nią polega na wykorzystywaniu udostępnionych interfejsów. Wywołanie odpowiedniej metody fabrykującej pozwala na stworzenie obiektu klasy *PluginManager*, który umożliwia przypisanie do siebie wszystkich istniejących pluginów z określonego źródła. Odwołanie się do określonego plugina zawartego w utworzonym menadżerze możliwe jest przy użyciu metody getPlugin. Kod umożliwiający wykonanie wyżej opisanych funkcjonalności zaprezentowano na listingu 3.

Listing 3 Główne metody umożliwiające dostęp do istniejących pluginów.

```
//tworzenie obiektu managera pluginów
  PluginManager pm = PluginManagerFactory.createPluginManager();
// dodawanie istniejących pluginów z wybranych lokalizacji
  pm.addPluginsFrom(new URI("classpath://*"));
  pm.addPluginsFrom(new File("plugins/").toURI());
  pm.addPluginsFrom(new File("plugin.jar").toURI());
  pm.addPluginsFrom(new URI("http://sample.com/plugin.jar"));
  pm.addPluginsFrom(new ClassURI(ServiceImpl.class).toURI());
// uzyskanie dostępu do określonego plugina za pośrednictwem managera
  Plugin p = pm.getPlugin(Plugin.class);
```

Tworząc pomocniczy obiekt klasy *PluginManagerUtil* istnieje możliwość uzyskania wszystkich zarejestrowanych pluginów w postaci kolekcji obiektów. Odbywa się to za pomocą metody getPlugins zaprezentowanej na listingu 4.

Listing 4 Sposób uzyskiwania kolekcji pluginów.

```
Collection<Plugin> pCol;
PluginManagerUtil pmu = new PluginManagerUtil(pm);
pCol = pmu.getPlugins(Plugins.class);
```

Ponadto w celu przyspieszenia implementacji nowych wtyczek JSPF wyposażony został również w szereg adnotacji takich jak:

- **@PluginImplementation** – adnotacja używana w celu oznaczenia klasy, którą system powinien traktować jako plugin. Klasa ta powinna implementować również interfejs dziedziczący po dostarczony przez JSPF interfejsie *Plugin*. Poniżej zaprezentowano listingu 5 wykorzystujący opisywaną adnotację,

Listing 5 Przykład zastosowania adnotacji @PluginImplementation.

```
@PluginImplementation
public class PluginImpl implements SamplePlugin {
     ...
}
```

- **@IncjectPlugin** – zastosowanie tej adnotacji oznacza, że pole nią oznaczone powinno być reprezentowane przez interfejs określonej wtyczki. W przypadku, gdy podany interfejs jest niedostępny pole to przyjmuje wartość *null*. Sposób zastosowania opisywanej adnotacji zaprezentowano na listingu 6,

Listing 6 Przykład zastosowania adnotacji @InjectPlugin.

```
@InjectPlugin
public SomePlugin requariedPlugin;
```

- **@PluginLoaded** – adnotacja umożliwiająca oznaczenie wybranych metody publicznych w celu wywołania ich w chwili, gdy plugin zdefiniowany w owej metodzie zostanie poprawnie dodany do managera. Wykorzystanie opisywanej adnotacji przedstawia listing 7,

Listing 7 Przykład zastosowania adnotacji @PluginLoaded.

```
@PluginLoaded
public void checkPlugin(SomePlugin plugin){
    System.out.println("Plugin " + plugin + " został załadowany.");
}
```

- **@Init** – metoda publiczna oznaczona ta adnotacja, po poprawnym wczytaniu i zarejestrowaniu plugina w którym została zaimplementowana, będzie wywołana jako pierwsza. Sposób wykorzystania wyżej opisanej adnotacji przedstawiono na listingu 8,

Listing 8 Przykład zastosowania adnotacji @Init.

```
@Init
public void init(){
   System.out.println("Hello plugin");
}
```

- **@Timer** – adnotacja umożliwiająca oznaczenie wybranej metody publicznej w celu cyklicznego jej wykonywania. Odstępy czasu jakie powinny występować pomiędzy wywołaniem tej metody określa parametr *period*. Przykładowy kod prezentuje zastosowanie opisywanej adnotacji zaprezentowano na listingu 9,

Listing 9 Przykład zastosowania adnotacji @Timer.

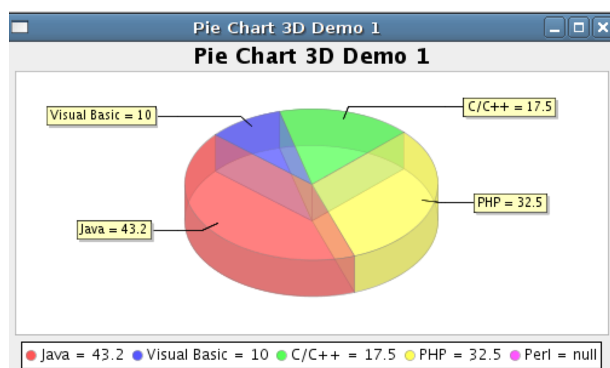
```
@Timer(period = 1000)                // interwał czasowy = 1 s
public void greet(){
    System.out.println("Hello again");
}
```

Możliwość pracy na udostępnionych interfejsach oraz adnotacji sprawia, że JSPF jest bardzo przydatnym narzędziem umożliwiającym przystosowanie wybranego systemu do pracy modułowej.

4.5. JFreeChart

W systemie informatycznym badającym efektywność linii produkcyjnych skorzystano również z biblioteki JFreeChart (patrz [10],[12]). Powstała ona w 1999 roku, jako projekt autorstwa Davida Gilberts'a, oparty na licencji LGPL. JFreeChart umożliwia generowanie wykresów będących graficzną reprezentacją dostarczonych wcześniej danych. Do podstawowych cech opisywanej biblioteki należą:

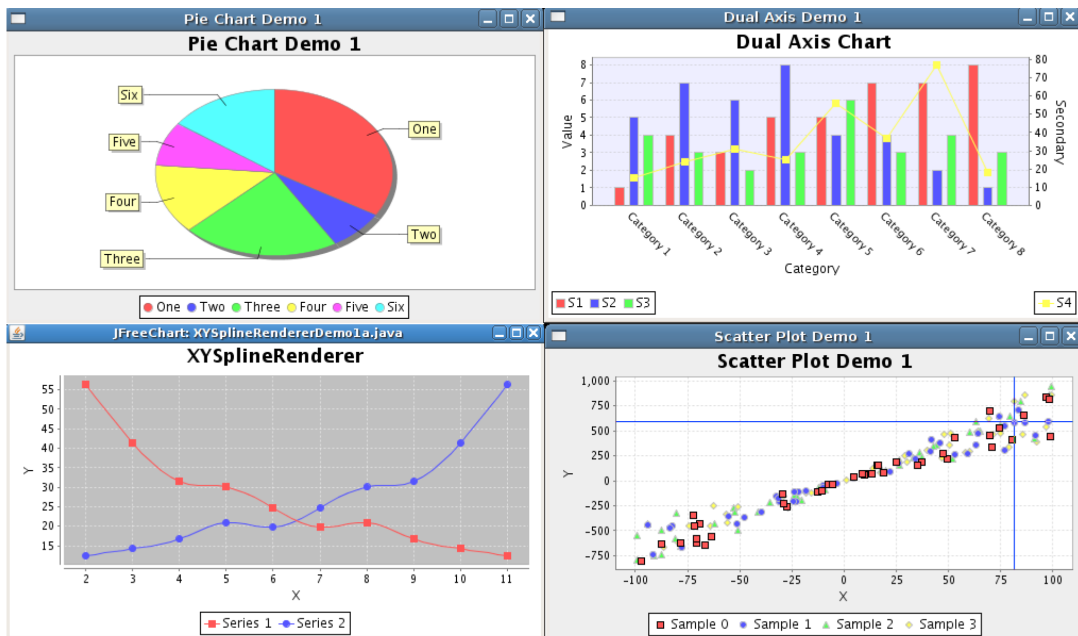
- dobrze udokumentowane API,
- możliwość wykorzystania biblioteki w aplikacjach desktopowych oraz webowych,
- generowanie tworzonych wykresów w formacie 2D i 3D. Zaprezentowany poniżej rysunek 21 przedstawia przykład wykresu 3D.



Rysunek 21 Wykres kołowy 3D - biblioteka JFreeChart.
Źródło: http://www.jfree.org/jfreechart/images/PieChart3DDemo1.png.

- wsparcie dla wielu typów wyjściowych takich jak: komponenty Swing, pliki graficzne (JPEG i PNG) czy pliki w formacie wektorowym (PDF, EPS, SVG),

- obsługa wielu typów wykresów. Poniższy rysunek 22 przedstawia niektóre z dostępnych typów wykresów.



Rysunek 22 Przykład dostępnych typów wykresów – JFreeChart.
Źródło: http://www.jfree.org/jfreechart/samples.html.

Przykładowy sposób implementacji umożliwiający wygenerowanie wykresu kołowego przedstawiono na listingu 10 oraz listingu 11.

Listing 10 Tworzenie zestawu danych dla wykresu kołowego - biblioteka JFreeChart.

```
DefaultPieDataset dane = new DefaultPieDataset();
dane.setValue("wartość 1", 25);
dane.setValue("wartość 2", 45);
dane.setValue("wartość 3", 30);
```

Dla określonego typu wykresu stworzono obiekt reprezentujący określony zestaw danych. W przypadku wykresu kołowego jest to obiekt klasy *DefaultPieDataset*. Tak przygotowany zestaw danych należy przekazać do obiektu tworzącego wymagany wykres.

Listing 11 Tworzenie obiektu reprezentującego wykres kołowy – JfreeChart.

```
JFreeChart wykres = ChartFactory.createPieChart(
                        "Wykres kołowy",          // tytuł wykresu
                        dane,                     // zestaw danych
                        true,                     // opis legendy
                        true,                     // tooltips
                        false
                        );
```

Obiekt klasy *JFreeChart* utworzono za pomocą metody fabrykującej pochodzącej z abstrakcyjnej klasy *ChartFactory*. W zależności od typu wykresu wymagane jest również dostarczenie odpowiednich argumentów zdefiniowanych w poszczególnych metodach fabrykujących.

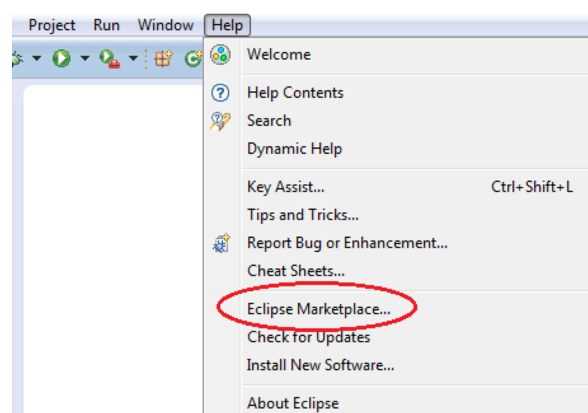
4.6. Eclipse IDE

W chwili obecnej jedną z najpopularniejszych platform wspomagającą tworzenie oprogramowania jest Eclipse (patrz [6],[11]). Inicjatorem jej stworzenia była firma IBM, która w 2001 roku rozpoczęła pracę nad budową zintegrowanego środowiska programistycznego. W 2004 roku podjęto decyzję o kontynuowaniu projektu pod patronatem utworzonej specjalnie do tego celu fundacji zrzeszającej firmy i organizacje zainteresowane rozwojem platformy już w ramach projektu *open source*.

Eclipse, jako zintegrowane środowisko programistyczne, w przewarzającej mierze zostało napisane przy użyciu języka Java, a jej podstawowa instalacja umożliwia tworzenie oprogramowania właśnie w tym języku. Nie oznacza to jednak, że na tym kończą się możliwości tego narzędzia. Dzięki specyficznej konstrukcji, której integralną częścią jest silnik zarządzający i definiujący zależnościami pomiędzy rozszerzeniami, platforma ta umożliwia znaczną jej rozbudowę. Korzystając z szerokiej gamy udostępnionych przez producenta wtyczek można w łatwy sposób dostosowywać możliwości tego środowiska do indywidualnych potrzeb. Powoduje to, że platforma staje się uniwersalnym narzędziem umożliwiającym między innymi.:

- dostosowanie platformy do określonego języka programowania tj. C++, PHP, Python, Perl, Ruby,
- tworzenie GUI przy użyciu edytora,
- współpracę z serwerami baz danych,
- modelowanie baz danych z możliwością generowania kodu aplikacji opartego na zaprojektowanym diagramie,
- współpracę z serwerami aplikacji tj. Tomcat, Glassfish, JBoss,
- usprawnienie pracy zespołowej poprzez pluginy pozwalające na kontrolę wersji oraz zarządzanie projektem,
- tworzenie dokumentacji projektowej.

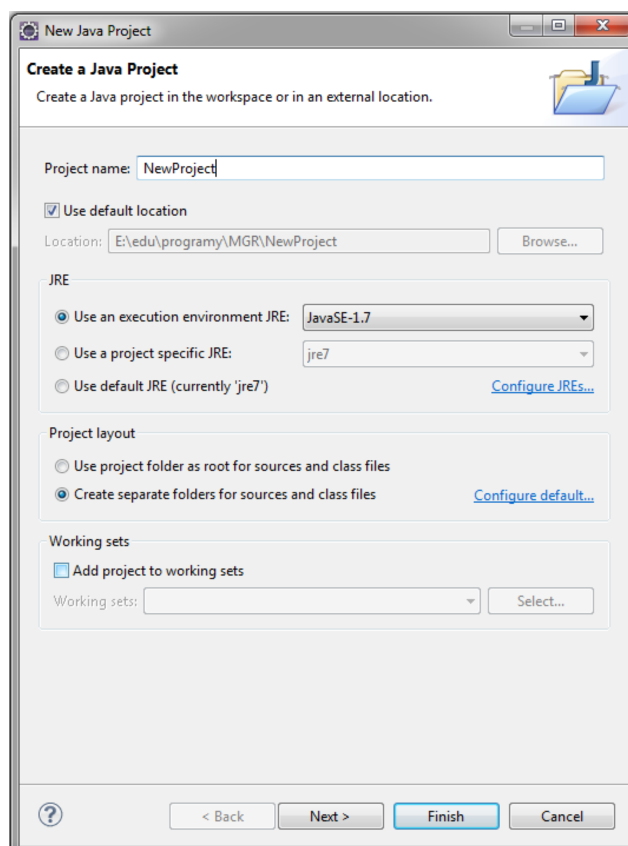
Samo wyszukiwanie i instalowanie pluginów możliwe jest między innymi poprzez tzw. Eclipse Marketplace, czyli jedną z funkcji opisywanego środowiska zaprezentowanego na rysunku 23.



Rysunek 23 Eclipse Marketplace. Źródło: Opracowanie własne

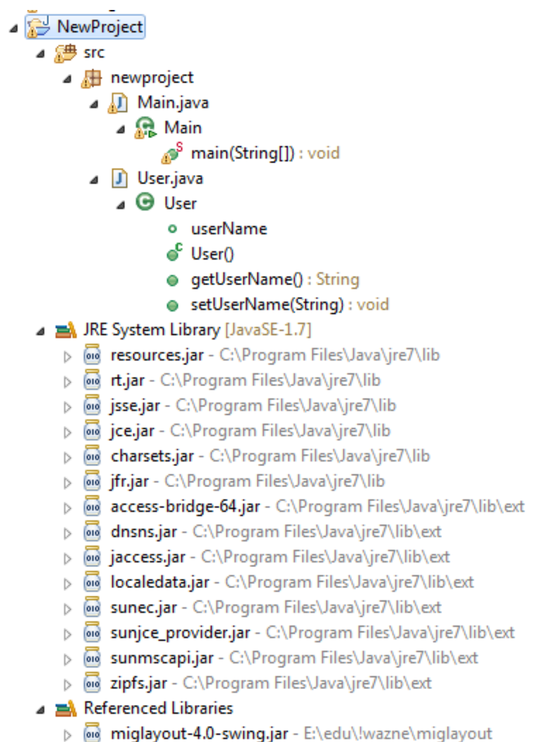
Tworzenie nowego projektu polega na wybraniu z menu *File* > *New* > *Project* lub użyciu skrótu klawiszowego *Alt+Shift+N*. Wyświetlona zostaje wówczas lista projektów jakie mogą zostać stworzone przy użyciu opisywanego środowiska. Wybranie jednego z nich powoduje wyświetlenie okna, w którym należy określić szczegóły dotyczące tworzonego projektu. System

zaimplementowany na potrzeby tej pracy, został stworzony na bazie *Java Project*, którego kreator wymaga określenia między innymi jego nazwy i środowiska uruchomieniowego. Sposób tworzenia nowego projektu zaprezentowano na rysunku 24.



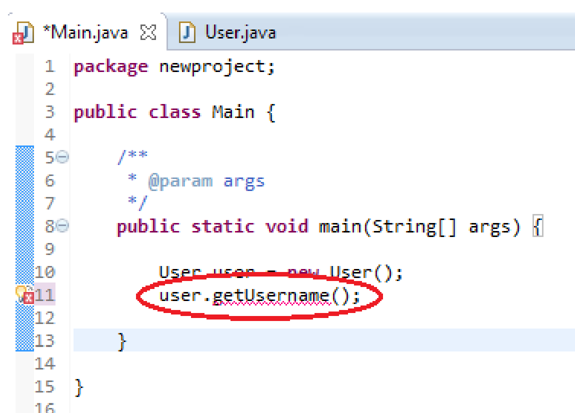
Rysunek 24 Tworzenie nowego projektu - Eclipse Juno. Źródło: Opracowanie własne

Naciśnięcie przycisku *Next* umożliwia zdefiniowanie dodatkowych ustawień, takich jak zewnętrzne biblioteki, natomiast przycisk *Finish* kończy pracę kreatora projektu, czego efektem jest wygenerowanie struktury katalogów dla nowego projektu. Hierarchiczna przeglądarka dostępna w opisywanym środowisku umożliwia w bardzo łatwy sposób poruszanie się po strukturze projektu, dając dostęp do stworzonych w projekcie klas, zaimplementowanych w obrębie tych klas pól, konstruktorów oraz metod, czy użytych w projekcie zewnętrznych bibliotek. Przykład takiej struktury zaprezentowano na rysunku 25.



Rysunek 25 Przykładowa struktura katalogowa dla projektu Java – Eclipse Juno. Źródło: Opracowanie własne

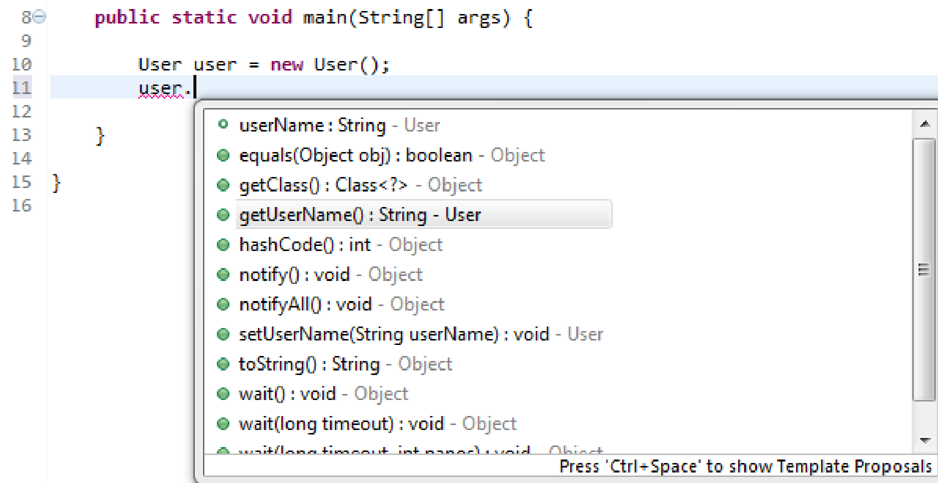
Pisanie kodu za pomocą Eclipse ułatwia wbudowany w platformę Asystent wprowadzenia (*code assist*), dzięki któremu składnia pisana przez programistę jest na bieżąco sprawdzana, a potencjalne błędy zostają odpowiednio oznaczone. Przykładowy sposób informowania przez Eclipse o nieprawidłowościach w implementacji zaprezentowano na rysunku 26.



Rysunek 26 Wskazanie potencjalnego błędu w implementacji - Eclipse Juno. Źródło: Opracowanie własne

Asystent wprowadzenia wspomaga również samą implementację kodu poprzez prezentowanie na bieżąco listy zawierającej dostępne pola i metody dla obiektu, do którego programista aktualnie się odwołuje. Opisywana funkcjonalność została przedstawiona na rysunku 27. Pozwala ona na szybsze wprowadzanie kodu, unikanie błędów związanych z

nieprawidłowym wprowadzeniem nazw pól lub metod oraz łatwą i szybką prezentację ewentualnych możliwości implementacyjnych.



Rysunek 27 Asystent wprowadzenia - Eclipse Juno. Źródło: Opracowanie własne

Szeroka możliwość rozbudowy platformy o nowe funkcjonalności, dostępność dla większości systemów operacyjnych, łatwy i przejrzysty interfejs użytkownika oraz szereg wbudowanych udogodnień sprawiają, że Eclipse na chwile obecną jest jedną z najchętniej stosowanych platform wspomagających tworzenie oprogramowania. Dostęp do niej, na zasadach otwartego oprogramowania opartego na specjalnie stworzonej licencji EPL (*Eclipse Public License*) sprawia, że jest ona systematycznie rozwijana, a udział w jego rozbudowie może mieć praktycznie każdy.

4.7. Baza danych mySQL

System zarządzania bazami danych o nazwie mySQL (patrz [15],[17]) został stworzony i udostępniony na zasadach *open source* przez firmę Oracle. Jest on oparty na relacyjnym modelu danych, w którym dane przechowywane są w tabelach. System ten charakteryzuje się dużą szybkością działania, niezawodnością oraz skalowalnością.

Komunikacja klient/serwer możliwa jest dzięki udostępnionym przez producenta API (*Application Programming Interface*), dedykowanym dla aplikacji klienckich zaimplementowanych w określonych językach programowania. W zależności od środowiska w jakim dana aplikacja została napisana, połączenie z bazą danych mySQL użytkownik, może uzyskać wspomagając min. takimi narzędziami jak:

- **Connector/ODBC** – sterownik zapewniający dostęp do bazy mySQL za pomocą standardowego interfejsu ODBC (*Open Database Connectivit*),
- **C API** – biblioteka umożliwiająca komunikowanie się aplikacji klienckie napisanej w języku C z bazą mySQL,
- **Connector/C++** – sterownik przeznaczony dla aplikacji klienckich napisanych w języku C++,
- **Connector/J** – sterownik zapewniający komunikację serwera mySQL z aplikacją kliencką przy użyciu JDBC (*Java Database Connectivity*),
- **Connector/Net** – sterownik przeznaczony dla aplikacji klienckich napisanych w języku .NET,

- **Connector/Python** – sterownik przeznaczony dla aplikacji klienckich napisanych w języku Python,
- **MySQL/Ruby API** – biblioteka do komunikacji klient/serwer przeznaczona dla języka Ruby.

Kolejną dużą zaletą opisywanego systemu jest, że z powodzeniem można go stosować w większości znanych systemów operacyjnych takich jak Microsoft Windows, Linux, Mac OS X, FeeBSD, HP-UX czy Solaris.

W poniższej tabeli [Tabela 1] zaprezentowano najważniejsze typy danych występujące w bazach danych mySQL w rozbiciu na trzy kategorie.

Tabela 1Wybrane typy danych w systemie mySQL. Źródło: Opracowanie własne

Typy numeryczne			
Kategoria	**Typ danych**	**Zakres** [3]	**Rozmiar**
Liczby całkowite	TINYINT [(M)]	od -128 do 127 *od 0 do 255*	1 bajt
	BIT, BOOL,BOOLEAN	Tożsame z TINYINT(1)	
	SMALLINT[(M)]	od -32768 do 32767 *od 0 do 65535*	2 bajty
	MEDIUMINT[(M)]	od -8388608 do 8388607 *od 0 do 16777215*	3 bajty
	INT[(M)], INTEGER[(M)]	od -2147483648 do 2147483647 *od 0 do 4294967295*	4 bajty
	BIGINT[(M)]	od -9223372036854775808 do 9223372036854775807 *od 0 do 18446744073709551615*	8 bajty
Liczby zmiennoprzecinkowe	FLOAT[(M,D)]	od -3.402823466E+38 do-1.175494351E-38, 0, *od 1.175494351E-38* *do 3.402823466E+38*	4 bajty
	DOUBLE[(M,D)]	od -1.7976931348623157E+308 do -2.2250738585072014E-308, 0, *od 2.2250738585072014E-308 do* *1.7976931348623157E+308*	8 bajty
	DOUBLE PPRECISION[(M,D)]		
	REAL[(M,D)]		
Liczby stałoprzecinkowe	DECIMAL[(M,[D])]		(M+2) bajtów
	DEC[(M,[D])]		
	NUMERIC[(M,[D])]		
	FIXED[(M,[D])]		
Typy dat i czasu			
Typ danych	**Zakres**	**Format**	**Rozmiar**
DATE	od ‘1000-01-01’ do ‘9999-12-31’	‘YYYY-MM-DD’	3 bajty
DATETIME	od ‘1000-01-01 00:00:00’ do ‘9999-12-31 23:59:59’	‘YYYY-MM-DD HH:MM:SS’	8 bajty
TIMESTAMP	od ‘1970-01-01 00:00:01’ do	‘YYYY-MM-DD HH:MM:SS’	4 bajty

[3] Zapis czcionką pochyłą reprezentuje wartości z argumentem UNSIGNED

	'2038-01-19 03:14:07'		
TIME	od '-838:59:59' do '838:59:59	'HH:MM:SS'	3 bajty
Typy łańcuchowe			
Typ danych	**Zakres**		**Rozmiar**
CHAR(M)	od 0 do 255		M bajtów
VARCHAR(M)			L+1 bajtów
TINTBLOB, TINTTEXT			
BLOB, TEXT	od 0 do 65535		L+2 bajtów
MEDIUMBLOB, MEDIUMTEXT	od 0 do 16777215		L+3 bajtów
LONGBLOB, LONGTEXT	od 0 do 4294967295		L+4 bajtów

, gdzie:
M – maksymalny rozmiar wartości. Dopuszczalna maksymalna wartość wynosi 255.
D – określa liczbę cyfr dziesiętnych w liczbach zmiennoprzecinkowych i stałoprzecinkowych. Dopuszczalna maksymalna liczba cyfr po przecinku wynosi 30.
[] – wartość opcjonalna.
L – długość łańcucha tekstowego.

Dodatkowo istnieje możliwość zadeklarowania atrybutu *UNSIGNED* dla określonej wartości typu numerycznego w sytuacji, gdy wartości te nie mogą być liczbami ujemnymi, tym samym zyskując rozszerzenie górnego zakresu.

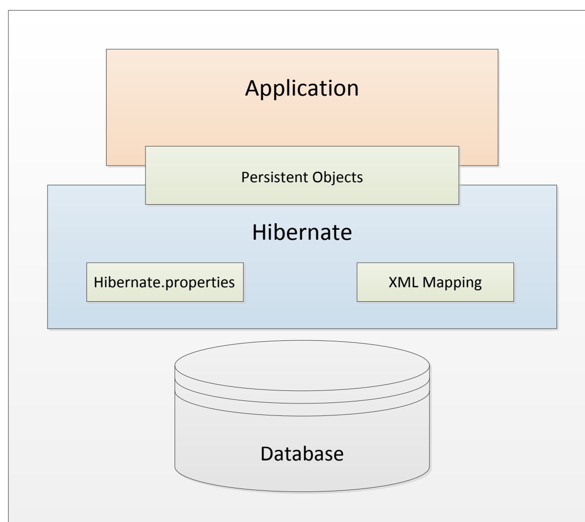
Operacje na bazach danych systemu mySQL wykonywane są przy użyciu deklaratywnego języka zapytań SQL (*Structured Query Language*). Oznacza to, że tworzone za jego pomocą zapytania zorientowane są na wynik, abstrahując od sposób ich wykonania. Formowane zapytania mogą należeć do jednej z 4 kategorii należących do SQL:

- **DML** (*Data Manipulation Language*) – do kategorii tej należą takie polecania jak INSERT, UPDATE oraz DELETE. Polecenia te służą do wstawiania, modyfikowania i usuwania danych w bazie,
- **DDL** (*Data Definition Language*) – kategoria języka SQL zawierające polecenia odpowiedzialne za operacje na takich strukturach tabele, indeksy, perspektywy czy same bazy danych. Głównymi poleceniami należącymi do tej kategorii są CREATE, DROP, ALTER,
- **DCL** (*Data Control Language*) – część języka SQL obejmujące polecenia służące do zarządzania uprawnieniami do obiektów bazy. Do najważniejszych zaliczyć można GRANT, REVOKE, DENY,
- **DQL** (*Data Query Language*) – zawiera tylko jedno polecenie – SELECT. Służy ono do wyszukiwania danych w bazie.

4.8. Hibernate

Hibernate (patrz [18]) jest biblioteką udostępnioną na zasadach *open source*, której w 2001 roku inicjatorem był Gavin King. Jest ona technologią umożliwiającą obiektowo-relacyjne (*Object-Relational Mapping,* ORM) mapowanie danych. Za jej pomocą obiekty tworzone w języku Java odwzorowywane zostają na postać relacyjnych danych, umożliwiając w ten sposób

kompatybilną współpracę z bazami danych. Stanowi ona więc warstwę pośredniczącą (*persistance layout*) pomiędzy określonym systemem bazy danych, a aplikacją kliencką. Ogólną architekturę opisywanej biblioteki zaprezentowano na rysunku 28.



Rysunek 28 Ogólna architektura Hibernate. Źródło: Opracowanie własne

W skład biblioteki Hibernate wchodzi kilka mniejszych modułów pełniących następujące funkcje:

- **Hibernate Core** – moduł ten stanowi integralną część biblioteki Hibernate, na podstawie którego utworzone zostały kolejne moduły opisywane poniżej. Umożliwia zdefiniowanie sposobu mapowania obiektów przy użyciu dokumentów XML. Przykładowy kod prezentujący sposób takiego odwzorowania zaprezentowano poniżej zaprezentowano na listingu 12,

Listing 12 Przykład mapowania obiektów zdefiniowana w pliku XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
          "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="s9821.app.model">

   <class name="User" table="USER">

      <id name="userId" column="ID_USER" type="int">
         <generator class="increment"/>
      </id>

      <property name="firstName" column="first_name" type="string" />
      <property name="lastName" column="last_name" type="string" />

   </class>
</hibernate-mapping>
```

- **Hibernate Annotations** – moduł będący rozszerzeniem Hibernate Core, pozwalający na stosowanie adnotacji w celu mapowania tworzonych obiektów. Przykład prezentujący zastosowanie adnotacji przedstawiono na listingu 13,

Listing 13.Sposób mapowania obiektów za pomocą adnotacji.

```
@Entity
```

```
@Table(name="USER")
public class User {
  @Id
  @GeneratedValue
  @Column(name="ID_USER")
  private int userId;

  @ Column(name="first_name")
  private String firstName;
  @ Column(name="last_name")
  private String lastName;


   //konstruktor i zestaw getterów i seterów
   ...
}
```

- **Hibernate EntityManager** – rozszerzenie Hibernate Core udostępniające klasę *EntityManager* z poziomu której możliwa jest komunikacja z bazą danych. W połączeniu z Hibernate Core pozwala na mapowanie obiektowo-relacyjne zgodne ze standardem *Java Persistance API*,
- **Hibernate Sharads** – rozszerzenie Hibernate Core ułatwiające pracę z wieloma bazami danych,
- **Hibernate Validator** – moduł rozszerzający ilość dostępnych adnotacji umożliwiając w ten sposób stosowanie dodatkowych ograniczeń na pola mapowanego obiektu. W tabeli [Tabela 2] zaprezentowanej poniżej opisano niektóre z wybranych adnotacji dostępnych poprzez Hibernate Validator,

Tabela 2 Wybrane typy adnotacji - Hibernate Validator. Źródło: Opracowanie własne

Rodzaj adnotacji	Wymagany typ danych	Zastosowanie
@DecimalMax	BigDecimal, BigInteger, String, byte, short, int, long	Wartość pola musi być mniejsza lub równa od wartości zadeklarowanej przez tą adnotację
@DecimalMin	BigDecimal, BigInteger, String, byte, short, int, long	Wartość pola musi być większa lub równa od wartości zadeklarowanej przez tą adnotację
@Email	String	Sprawdza czy zadeklarowana wartość String jest adresem e-mail
@Future	java.util.Date, java.util.Calendar	Sprawdza czy zadeklarowana wartość jest datą przyszłą
@Past	java.util.Date, java.util.Calendar	Sprawdza czy zadeklarowana wartość jest datą przeszłą
@NotNull	wszystkie typy danych	Sprawdza czy pole lub właściwość nie jest wartością null
@NotEmpty	String	Sprawdza czy pole lub właściwość nie jest wartością pustą lub null
@Pattern(regex=, flag=)	String	Sprawdza czy podana wartość pasuje do określonego w parametrze wyrażenia regularnego

- **Hibernate Search –** umożliwia pełnotekstowe wyszukiwanie przy użyciu biblioteki Lucene,
- **Hibernate Tools –** moduł udostępniający zestaw narzędzi wspomagający pracę z Hibernate Core,
- **NHibernate –** translacja biblioteki Hibernate dla platformy .NET.

Poprawne zaimplementowanie biblioteki Hibernate rozpocząć należy od stworzenia pliku konfiguracyjnego (domyślnie jest to „hibernate.cfg.xml”). Pozwala on na zdefiniowanie parametrów potrzebnych do właściwego działania biblioteki w odniesieniu do zastosowanej bazy danych. Do najważniejszych własności zdefiniowanych w pliku konfiguracyjnym należą:

- **connection.driver_class** – określa klasę sterownika JDBC dla określonej bazy danych,
- **connection.url** – określa adres połączenia z bazą danych,
- **connection.username** – login dostępu do bazy danych,
- **connection.password** – hasło dostępu do bazy danych,
- **connection.pool_size** – ilość jednoczesnych połączeń z bazą danych,
- **dialcet** – rodzaj dialektu używanego w określonej bazie danych,
- **show_sql** – wartość *true* umożliwia podgląd tworzonych zapytań bazodanowych,
- **hbm2ddl.auto** – w zależności od zastosowanego parametru umożliwia odpowiedzenie działanie na strukturę bazy danych. Dostępne parametry to: *create*, *create-drop*, *update*, *validate*.

Mapowanie obiektów na postać relacyjną polega na odwzorowaniu „klas trwałych” zaimplementowanych w taki sposób, aby spełniały reguły POJO (*Plain Old Java Object*). Klasy te posiadać muszą prywatne pola opisujące cechy reprezentowanego obiektu oraz zestaw metod umożliwiających pośredni dostęp do tych pól. Przykładowy kod implementujący taką klasę przedstawiono na listingu 14.

Listing 14 Przykład klasy spełniającej reguły POJO

```
public class Employee {
   private int empID;
   private String firstName;
   private String lastName;
      public int getEmpID(){
         return empID;
      }
      public void setEmpID(int empID){
         this.empID = empID;
      }
      // analogiczne metody dla pozostałych pól
      ...
}
```

Poprzez zastosowanie odpowiednich adnotacji lub plików XML, Hibernate umożliwia odwzorowanie zachodzących pomiędzy poszczególnymi obiektami, takich związków asocjacyjnych jak:

- jeden do jednego (1:1),
- jeden do wielu (1:N),
- wiele do jednego (N:1),
- wiele do wielu (N:M).

Wykonywanie określonych zapytań do bazy danych odbywa się przy użyciu obiektowego języka HQL (*Hibernate Query Language*), będącego pochodną języka SQL. Zapytania tworzone za jego pomocą przekładane są automatycznie na język SQL, a jego wynik zwracany jest w formie referencji do określonego obiektu. Zestawienie połączenia oraz wykonywanie określonych operacji w bazie danych zapewniają dostarczone przez Hibernate wymienione poniżej interfejsy:

- ***Configuration*** – obiekt tworzony przy użyciu tego interfejsu umożliwia odpowiednie skonfigurowanie Hibernate w odniesieniu do zastosowanej bazy danych. Pozwala na odniesienie się do zdefiniowanego wcześniej pliku konfiguracyjnego,
- ***SessionFactory*** – obiekt umożliwiający wielowątkową pracę w obrębie danej aplikacji. Pozwala na tworzenie obiektów klasy *Session*, za pomocą których realizowane są określone operacje na bazach danych. Ilość obiektów *SessionFactory* uzależniona jest od ilości baz danych z których aplikacja korzysta. Z reguły dla jednej bazy danych tworzy się jeden obiekt *SessionFactory,*
- ***Session*** – obiekt tworzony dla pojedynczego procesu wykonywanego na bazie danych. Powstaje przy użyciu *SessionFactory* i zapewnia dostęp do podstawowych operacji na bazie danych. Koszt tworzenia i niszczenia obiektu jest niewielki, dlatego w obrębie jednej aplikacji może istnieć wiele takich obiektów,
- ***Transaction*** – obiekt tworzony przy użyciu tego interfejsu pozwala na definiowanie transakcji Hibernate w obrębie określonej sesji. Jedna sesja może obejmować wiele transakcji,
- ***Query*** – umożliwia wykonywanie zapytań do określonej bazy danych. Formowanie zapytań odbywa się przy użyciu języka HQL.

5. Implementacja systemu

Poniższy rozdział stanowi opis proponowanego sposobu implementacji generycznego systemu informatycznego umożliwiającego analizowanie efektywności linii produkcyjnych.

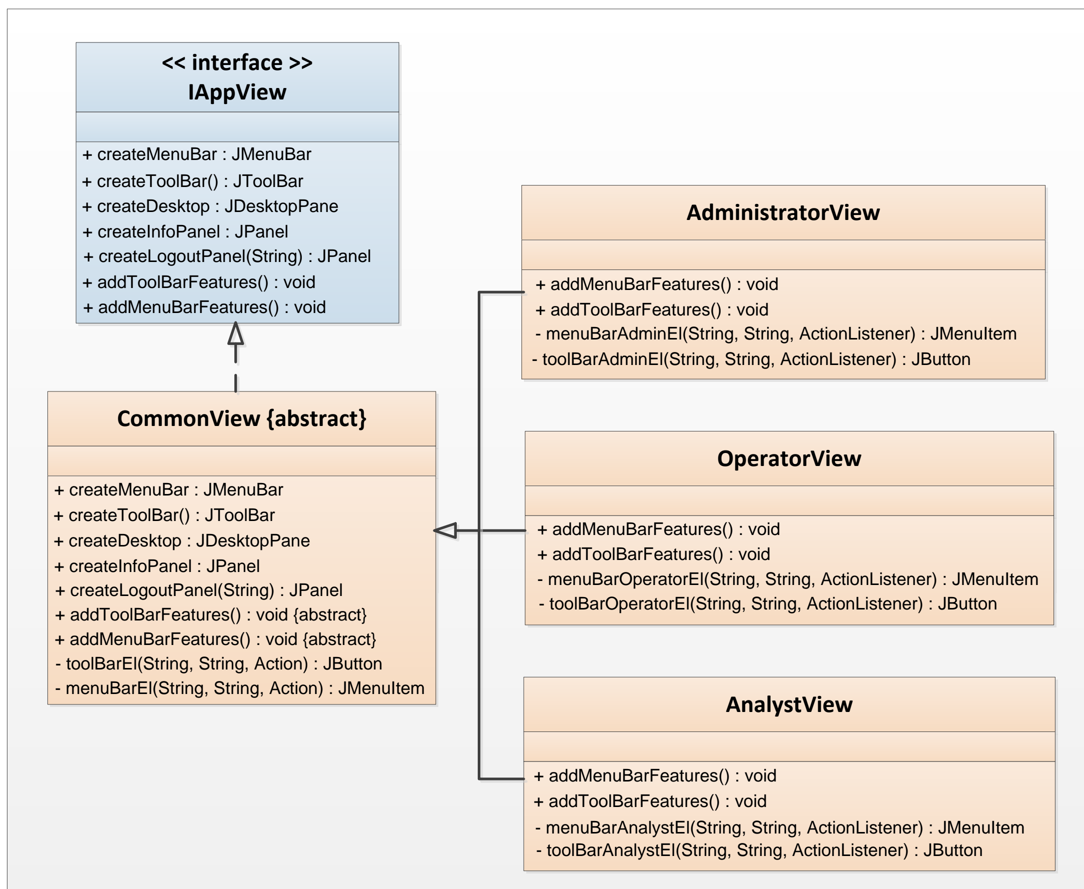
Konieczność sporządzania raportów będących źródłem danych polegających analizie powinna odbywać się regularnie w odstępie nie większym niż jedna zmiana, przy wykorzystaniu aplikacji dostępnej na miejscu pracy. Ponadto, z uwagi na konieczność przetwarzania dużej ilości danych pożądanym czynnikiem jest naturalne rozproszenie platformy. Z uwagi na opisane powyżej czynniki zdecydowano się na implementację systemu w technice desktopowej.

5.1. Program główny

Zaprojektowany prototyp systemu umożliwia użytkownikowi korzystanie z trzech odmiennych interfejsów udostępniających wybrane funkcjonalności. Dostęp do określonego interfejsu użytkownika (GUI), umożliwiono poprzez proces logowania z uwzględnieniem roli pełnionej w systemie. W zaprojektowanym prototypie użytkownicy mogą posiadać następujące role:

- **administrator** – rola pozwalająca użytkownikowi na zarządzanie wszystkim obiektami wchodzącymi w skład systemu. Posiada dostęp do funkcjonalności umożliwiającej definiowanie nowych linii produkcyjnych z uwzględnieniem maszyn oraz wytwarzanych na nich produktach. Pozwala również na zarządzanie użytkownikami systemu,
- **operator** – rola umożliwiająca sporządzanie raportów na podstawie których dokonywana jest analiza efektywności,
- **analityk** – rola udostępniająca funkcjonalności związane z przeprowadzaniem analiz efektywności linii produkcyjnych.

Proces tworzenia GUI odbywa się za pośrednictwem metod zadeklarowanych w *IAppView*. Jest to interfejs implementowany przez abstrakcyjną klasę *CommonView*, którą to z kolei dziedziczą trzy inne klasy o nazwach *AdministratorView*, *OperatorView, AnalystView*. Diagram UML prezentujący opisane powyżej zależności zaprezentowano na rysunku 29. Wspólne dla całego systemu komponenty GUI utworzone zostaną za pomocą metod wywołanych w klasie *CommonView*, natomiast komponenty przewidziane dla określonych ról zdefiniowano w klasach dziedziczących.



Rysunek 29 Diagram UML reprezentujący GUI systemu. Źródło: Opracowanie własne

Uzyskanie dostępu do GUI przeznaczonego dla określonej roli umożliwiono poprzez stworzenie metody fabrykującej getRole znajdującej się w klasie *ViewFactory*. Przy pomocy przekazanego parametru metoda ta zwraca odpowiedni obiekt klasy implementującej interfejs *IAppView*. Kod prezentujący metodę fabrykującą przedstawiono na listingu 15.

Listing 15 Sposób implementacji metody fabrykującej getRole

```
public static IAppView getRole(String role, LoginDetails user){
     if(role.equals("Administrator")){
        return new AdministratorView(user);
     }else if(role.equals("Operator")){
        return new OperatorView(user);
     }else if(role.equals("Analyst")){
        return new AnalystView(user);
     }
}
```

5.2. Mapowanie obiektowo-relacyjne

Zastosowanie oraz odpowiednie skonfigurowanie zewnętrznej biblioteki Hibernate pozwoliło na stworzenie warstwy, której zadaniem jest realizacja dostępu do danych oraz zapewnienie właściwej translacji obiektowo-relacyjnej.

Zestawienie połączenia pomiędzy aplikacją kliencką, a bazą danych zrealizowano poprzez odpowiednie zdefiniowanie pliku konfiguracyjnego „hibernate.cf.xml”, w sposób zaprezentowany na listingu 16.

Listing 16 Zestaw zdefiniowanych własności pliku konfiguracyjnego hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>

    <session-factory>

        <property name="connection.driver_class">
                 com.mysql.jdbc.Driver
        </property>

        <property name="connection.url">
                 jdbc:mysql://localhost:3306/mgr_s9821
        </property>

        <property name="connection.username">root</property>

        <property name="connection.password">s9821</property>

        <property name="connection.pool_size">2</property>
        <property name="dialect">
                 org.hibernate.dialect.MySQL5Dialect
        </property>

        <property name="show_sql">true</property>

        <property name="hbm2ddl.auto">update</property>

    </session-factory>

</hibernate-configuration>
```

Zdefiniowane w pliku własności pozwoliły na prawidłowe wykonywanie połączeń oraz operacji niezbędnych do prawidłowego funkcjonowania systemu. Zawarto w nim następujące informacje:

- nazwę klasy sterownika JDBC dla wybranej bazy danych,
- adres URL bazy danych,
- login oraz hasło do bazy danych,
- ilość jednoczesnych połączeń,
- rodzaj dialektu użytego w bazie danych.

Tworzenie obiektów reprezentujących trwałe dane zrealizowano poprzez zdefiniowanie odpowiednich klas POJO. W celu zapewnienia właściwego mapowania obiektowo-relacyjnego, w każdej z tych klas zastosowano szereg adnotacji udostępnionych przez bibliotekę Hibernate. Przykład implementacji klasy POJO reprezentującej jedną z tabel relacyjnej bazy danych przedstawiono na listingu 17.

Listing 17 Przykładowa implementacja klasy POJO z wykorzystaniem adnotacji

```
@Entity
public class ProductionLine {

 private int IDProductionLine;
 private String productionLineName;
 private String description;
 private List<Machine> machines = new ArrayList<Machine>();
```

```
 @Id
 @TableGenerator(name="idProductionLine",table="nextIDProductionLine",
               pkColumnName="productionLineKey",
               pkColumnValue="productionLineValue", allocationSize=1)
 @GeneratedValue(strategy=GenerationType.TABLE, generator="idProductionLine")
 public int getIDProductionLine() {
    return IDProductionLine;
 }

 public void setIDProductionLine(int iDProductionLine) {
    IDProductionLine = iDProductionLine;
 }
 // pozostałe gettery i settery
     ...
}
```

Każda z klas oznaczonych adnotacją *@Entity* posiada odwzorowanie w postaci tabeli w relacyjnej bazie danych. Domyślnie tabela przyjmuje nazwę danej klasy, jednak zastosowanie adnotacji *@Table* umożliwia zmianę nazewnictwa.

Pola zdefiniowane w takiej klasie odpowiadają poszczególnym kolumnom tabeli. W celu oznaczenia pola, który pełnić będzie funkcje klucza głównego zastosowano adnotację *@Id*. Biblioteka Hibernate umożliwia również wybór strategii określającej sposób generowania takiego klucza. W przedstawionym powyżej przykładzie, używając adnotacji *@GeneratedValue* skorzystano ze strategii opartej na tabeli. Zadaniem takiej tabeli jest przechowywanie pojedynczego rekordu reprezentującego wartość następnego możliwego do wykorzystania klucza głównego. W ten sposób generowanie wartości klucza głównego dla nowych rekordów tabeli realizowane jest automatycznie przez system z zachowaniem odpowiedniej kolejności.

Adnotacje udostępnione przez bibliotekę Hibernate umożliwiły również prawidłowe zdefiniowanie związków asocjacyjnych zachodzących pomiędzy poszczególnymi obiektami. Implementacja klas *User* oraz *LoginDetails* w zaprojektowanym prototypie stanowią przykład zdefiniowania relacji typu jeden-do-jednego. Na przedstawionym poniżej listingu 18 przedstawiono jeden z możliwych sposobów opisania tego typu relacji.

Listing 18 Przykład implementacji relacji 1:1

```
@Entity
public class User {
  private LoginDetails loginDetails;
  ...
  @OneToOne(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
  @JoinColumn(name="idLoginDetails")
  public LoginDetails getLoginDetails() {
     return loginDetails;
  }
  ...
}
@Entity
public class LoginDetails {
  private int idLoginDetails;
  private User user;
  ...
  @OneToOne(cascade=CascadeType.ALL, mappedBy="loginDetails")
  public User getUser() {
    return user;
  }
  ...
```

```
}
```

Wykorzystanie w klasie *User* adnotacji @*OneToOne* przy definiowaniu pola typu *LoginDetails* pozwoliło na odpowiednie opisanie relacji zachodzących pomiędzy obiektami tych klas. Używając dodatkowej adnotacji @*JoinColumn* w przekazanym parametrze określono nazwę pola jednoznacznie identyfikującego obiekt klasy *LoginDetails*. Jednocześnie w klasie *LoginDetails* zdefiniowano pole reprezentujące obiekt klasy *User*. Poprzez zastosowanie adnotacji @*OneToOne* oraz podanie odpowiedniego parametru mapującego (mappedBy="loginDetails"), wskazano nazwę pola identyfikującego powiązany obiekt. Tak zdefiniowane relacje odwzorowane zostały w relacyjnej bazie danych poprzez utworzenie w tabeli *User* dodatkowej kolumny zawierające klucze obce odpowiadające kluczom głównym z tabeli *LoginDetails*.

Adnotację pozwalającą na opisanie relacji typu jeden-do-wielu lub wiele-do-jednego wykorzystano między innymi podczas implementacji klas *Machine* oraz *ProductionLine*. Każda istniejąca w systemie linia produkcyjna może się składać z wielu maszyn, a każda maszyna przypisana jest do określonej linii produkcyjnej. W tej sytuacji klasę tworzącą obiekty reprezentujące poszczególne linie produkcyjne wyposażono w dodatkowe pole zawierające kolekcję obiektów klasy *Machine*. W sposób analogiczny w klasie tworzących obiekty określające maszyny zdefiniowano pole odnoszące się do obiektu reprezentującego określoną linii produkcyjnej. W zaprezentowanym poniżej listingu 19 przedstawiono sposób takiej implementacji.

Listing 19 Przykład implementacji relacji 1:N oraz N:1

```
@Entity
public class Machine {
  private int IDMachine;
  private ProductionLine productionLine;
  ...
  @ManyToOne(cascade=CascadeType.ALL)
  @JoinColumn(name="IDProductionLine")
  public ProductionLine getProductionLine() {
     return productionLine;
  }
  ...
}
@Entity
public class ProductionLine {
  private int IDProductionLine;
  private List<Machine> machines = new ArrayList<Machine>();
  ...
  @OneToMany(targetEntity=Machine.class,mappedBy="productionLine",
            cascade=CascadeType.ALL, fetch = FetchType.LAZY)
  public List<Machine> getMachines() {
     return machines;
  }
  ...
}
```

W celu prawidłowego zdefiniowania klas, pomiędzy którymi zachodzą opisywane powyżej relacje skorzystano z adnotacji @*OneToMany* oraz @*ManyToOne*. W klasie *Machine* wykorzystano dodatkowo adnotację @*JoinColumn* wskazując jednoznacznie identyfikator obiektów klasy *ProductionLine*. Za pomocą adnotacji @*OneToMany* w klasie *ProductionLine* zdefiniowano dodatkowe argumenty pozwalające na określenie nazwy klasy obiektów w odniesieniu do których zachodzi owa relacja oraz wskazanie do mapowanego w nich pola.

Podobnie jak w przypadku relacji jeden-do-jednego, tak zdefiniowane zależności odwzorowane zostały w relacyjnej bazie danych w postaci dodatkowej kolumny utworzonej w tabeli *Machine* zawierającej zestaw kluczy obcych do tabeli *ProductionLine*.

W zaprojektowanym prototypie systemu wyróżnić można również związki asocjacyjne typu wiele-do-wielu zachodzące min. pomiędzy obiektami klasy *LoginDetails*, a obiektami klasy *Role*. W celu prawidłowego zdefiniowania takich relacji posłużono się adnotacją *@ManyToMany*, a przykładowy sposób jej implementacji zaprezentowano w listingu 20.

Listing 20 Sposób implementacji relacji N:M

```
@Entity
public class LoginDetails {
  private int idLoginDetails;
  private List<Role> rolesList = new ArrayList<Role>();
  ...
  @ManyToMany
  @JoinTable(name="LoginDetails_Role",
             joinColumns={@JoinColumn(name="idLoginDetails")},
             inverseJoinColumns={@JoinColumn(name="idRole")})
  public List<Role> getRolesList() {
     return rolesList;
  }
  ...
}
@Entity
public class Role {
  private int idRole;
  private String roleName;
  private List<LoginDetails> loginDetailsList =new ArrayList<LoginDetails>();
  ...
  @ManyToMany
  @JoinTable(name="LoginDetails_Role",
             joinColumns={@JoinColumn(name="idRole")},
             inverseJoinColumns={@JoinColumn(name="idLoginDetails")})
  public List<LoginDetails> getLoginDetailsList() {
     return loginDetailsList;
  }
  ...
}
```

W obu powiązanych ze sobą klasach zdefiniowano pola reprezentujące kolekcje obiektów pomiędzy którymi zachodzi opisywana relacja. Każde z takich pól opisano adnotacją *@ManyToMany* oraz dodatkowo zastosowano adnotację *@JoinTable*. Argumenty przekazane za pomocą adnotacji *@JoinTabel* umożliwiły utworzenie w bazie danych dodatkowej tabeli posiadającej kolumny zawierające reprezentujące klucze główne do tabel połączonych związkiem.

W celu wykonywania operacji na bazie danych, aplikacja kliencka wyposażona została w klasę o nazwie *DBQueriesImpl*. W konstruktorze tej klasy zawarto odwołania do obiektów tworzonych za pomocą interfejsów udostępnionych poprzez bibliotekę Hibernate, umożliwiających właściwe zestawienie połączenie oraz wykonywanie określonych zapytań na bazie danych. Przykład implementacji konstruktora w klasie *DBQueriesImpl* przedstawiono na listingu 21.

Listing 21 Przykład wykorzystania interfejsów biblioteki Hibernate

```
public class DBQueriesImpl implements IDBQueries {
```

```
  private AnnotationConfiguration config;
  private SessionFactory factory;

  public DBQueriesImpl(){

     config  = new AnnotationConfiguration();
     config.addAnnotatedClass(User.class);
     config.addAnnotatedClass(LoginDetails.class);

     ...

     config.configure("hibernate.cfg.xml");
     factory = config.buildSessionFactory();
  }
 ...
}
```

Ponadto w klasie *DBQueriesImpl* zaimplementowano szereg metod umożliwiających wykonywanie niezbędnych zapytań do bazy danych. Zapytania te zdefiniowane zostały przy użyciu języka HQL, a zaprezentowany poniżej kod (Listing 22) przedstawia przykładową metodę umożliwiającą pobranie z bazy danych, użytkowników posiadającego określone inicjały.

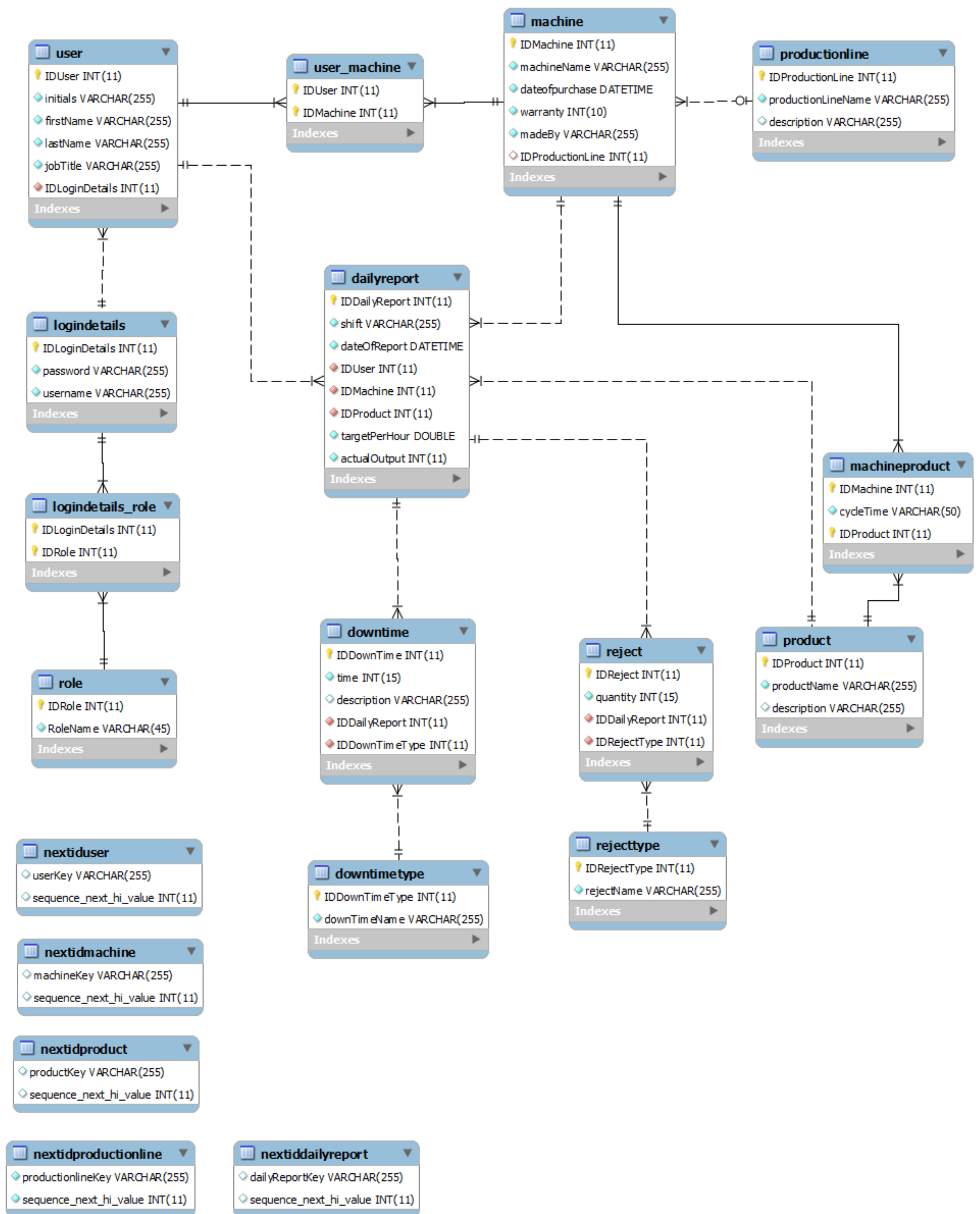
Listing 22 Przykład implementacji metody getSelectedUsers

```
public List<User> getSelectedUsers(ArrayList<String> usersInitials){

  User u = null;
  List<User> users = new ArrayList<User>();
  Transaction t = null;
  Session session = factory.openSession();

    try{
        for(String initial : usersInitials){
          t = session.beginTransaction();
          Query query = session.createQuery(
                        "from User where initials ='"+initial+"'");
          User u = (User) query.uniqueResult();
          users.add(u);
          t.commit();
        }
     }catch(HibernateException ex){
        if (t!=null) t.rollback();
        ex.printStackTrace();
     }
      return users;
}
```

Zaprezentowany poniżej rysunek 30 przedstawia schemat, zaprojektowanej w ramach stworzonego systemu bazy danych, odwzorowujący encji oraz zachodzących pomiędzy nimi związki asocjacyjne.



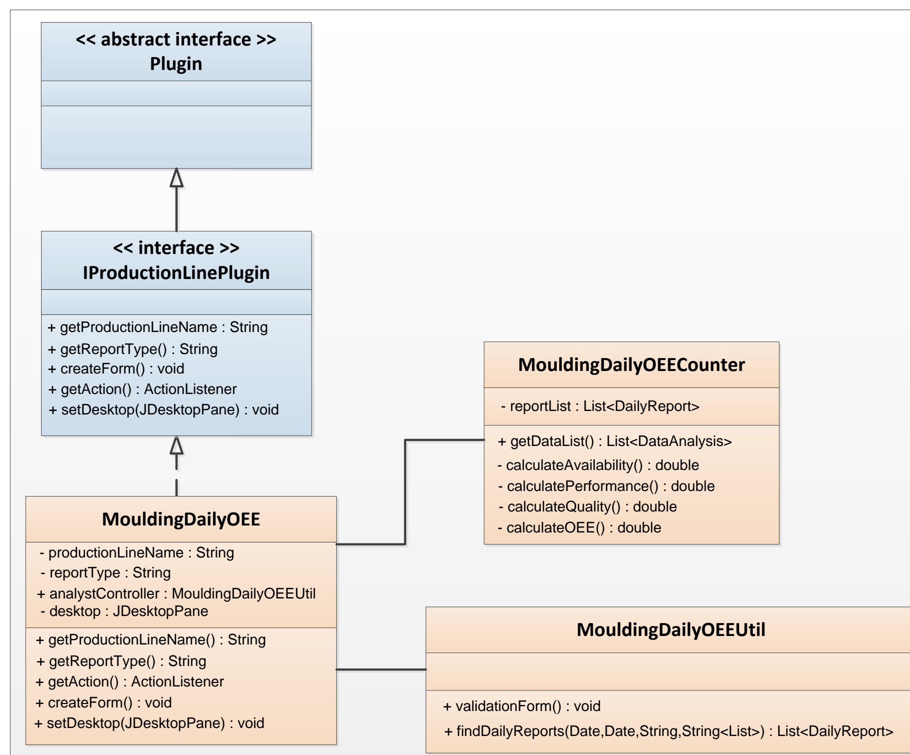
Rysunek 30 Schemat relacyjnej bazy danych systemu. Źródło: Opracowanie własne

5.3. System pluginów

Wspólnym mianownikiem charakteryzującym wszystkie linie produkcyjne jest rodzaj danych jakie należy uzyskać, aby przeprowadzić dla niej odpowiednią analizę efektywności. Dlatego też zaprojektowany system informatyczny wyposażono w funkcjonalność umożliwiającą sporządzanie raportów dotyczących przebiegu pracy urządzeń wchodzących w skład określonej linii produkcyjnej.

Ważnym czynnikiem wpływającym na obliczany współczynnik efektywności są również relacje jakie występują pomiędzy poszczególnymi składowymi linii produkcyjnych. Są to czynniki indywidualne charakterystyczne dla poszczególnych linii. Dlatego też, prototyp stworzonego systemu zakłada możliwość korzystania z pluginów, których celem jest odpowiednie zdefiniowanie takich zależności i uwzględnienia ich w trakcie przeprowadzania analizy.

Udostępnienie interfejsu o nazwie *IProductionLinePlugin* rozszerzającego, dostarczony przez JSPF, abstrakcyjny interfejs *Plugin* umożliwiło dostosowanie systemu do pracy modułowej. Mechanizm tworzenia kolejnych pluginów polega na stworzeniu klasy implementującej interfejs *IProductionLinePlugin*. Odpowiednie zdefiniowanie zawartych w takiej klasie metod umożliwiło uzyskanie w systemie dodatkowej funkcjonalności pozwalającej na przeprowadzenie analizy OEE dla wyszczególnionej linii produkcyjnej. Diagram UML przedstawiający opisywane interfejsy oraz klasy zaprezentowano na rysunku 31.



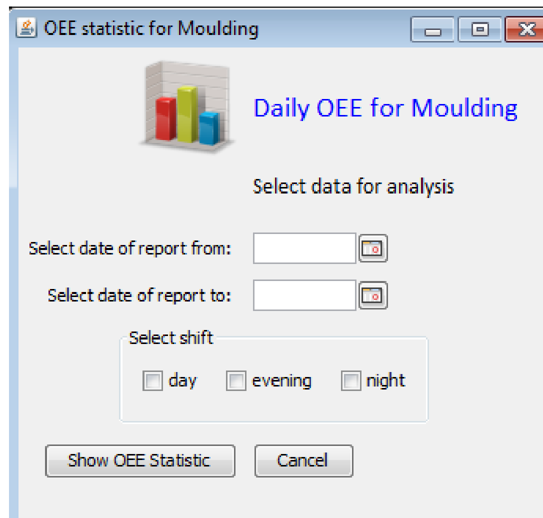
Rysunek 31 Diagram UML pluginu dla przykładowej linii produkcyjnej.
Źródło: Opracowanie własne

Metody zaimplementowane w przykładowej klasie *MouldingDailyOEE* mają za zadanie:

- ***getProductionLineName*** – metoda zwracająca nazwę linii produkcyjnej, którą dany plugin obsługuje,

* ***getReportType*** – metoda zwracająca nazwę przeprowadzanej analizy np.: „Daily OEE”,
* ***createForm*** – metoda, w której zawarto implementacje formularza zawierającego dane na podstawie których przeprowadzana jest analiza,
* ***getAction*** – metoda zwracające obiekt *ActionListener* przeznaczony do tworzenia formularza,
* ***setDesktop*** – metoda przekazująca w argumencie istniejący w systemie obiekt *JDesktopPanel*, wykorzystywany do wyświetlania formularzy oraz statystyk.

Przedstawiony na rysunku 32 formularz utworzony został za pomocą metody createForm. Wyposażono go w pola umożliwiające zdefiniowanie parametrów wymaganych do przeprowadzenia analizy.



Rysunek 32 Formularz do przeprowadzania analizy OEE. Źródło: Opracowanie własne

Wywołanie akcji przycisku generującego statystyki, powoduje utworzenie obiektu klasy *MouldingDailyOEEUtil*. Zawarta w tej klasie metoda validationForm ma za zadanie zweryfikowanie poprawności wypełnionego formularza. W przypadku pomyślnej weryfikacji, wykonywana jest metoda findDailyReports. Jej zadaniem jest pozyskanie z bazy danych kolekcji obiektów *DailyReport* spełniające wymagania zdefiniowane za pomocą formularza. Przykład implementacji opisywanej metody zaprezentowano na listingu 23.

Listing 23 Sposób implementacji metody findDailyReports

```
public List<DailyReport> findDailyReports(Date dateStart, Date dateEnd,
                  String prodLineName, String shift) throws ParseException {

  List<DailyReport> reports = null;
  Transaction t = null;
  Session session = factory.openSession();

  try{
     String dateDBStart = new SimpleDateFormat("yyyy-MM-dd").
                                                     format(dateStart);
     String dateDBEnd = new SimpleDateFormat("yyyy-MM-dd").format(dateEnd);
     t = session.beginTransaction();

     Query query = session.createQuery("from DailyReport dr where
                                      dr.dateOfReport >='"+dateDBStart+"'" +
                                      "and dr.dateOfReport <='"+dateDBEnd+
                                      "'and dr.shift = '"+shift+"'and
                                dr.machine.productionLine.productionLineName
```

```
                                   ='+prodLineName+"'"
                                  );

      reports = query.list();
      t.commit();

      }catch(HibernateException ex){
         if (t!=null) t.rollback();
         ex.printStackTrace();
      }
   return reports;
}
```

Na podstawie uzyskanych kolekcji tworzony jest nowy obiekt klasy *MouldingDailyOEECounter*. Posiada on metody, za pomocą których wyliczane są parametry dostępności, wydajności oraz jakości stanowiące składowe wyliczanego współczynnika efektywności. Obliczanie parametrów odbywa się w oparciu o dane dostarczone w uzyskanej wcześniej kolekcji raportów oraz znanych zależności występujących pomiędzy elementami poszczególnej linii produkcyjnej.

Przykładem może być obliczenie parametru dostępności, który wymaga uzyskania informacji dotyczących między innymi czasu poświęconego na nieplanowany postój. Listing 24 prezentuje sposób zliczania nieplanowanego przestoju dla linii produkcyjnej, której rozkład widoczny jest na rysunku 2. [4]

Listing 24 Przykładowy sposób implementacji metody calculateUnplannedDownTime.

```
private int calculateUnpannedDownTime(List<DailyReport> reports) {

   int unplannedDownTime = 0;

     for(DailyReport r : subReports){
       if(r.getMachine().getMachineName().equals("Maszyna B1") ||
          r.getMachine().getMachineName().equals("Maszyna B4")){

            List<DownTime> downTimeList = r.getDownTime();

              for(DownTime dt : downTimeList){

                String downTimeName = dt.getDownTimeType().getDownTimeName();

                if(downTimeName.equals("Material out of stock") ||
                   downTimeName.equals("Machine set-up by operator") ||
                   downTimeName.equals("Machine set-up by engineer") ||
                   downTimeName.equals("Waiting for documentation")||
                   downTimeName.equals("Waiting for tech. support") ||
                   downTimeName.equals("Job changeover")||
                   downTimeName.equals("Machine down") ||
                   downTimeName.equals("Temperature too high")){

                      unPlannedDownTime = unPlannedDownTime + dt.getTime();

                }
              }
        }
     }

   return unPlannedDownTime;
}
```

Zaimplementowana metoda, na podstawie listy dostarczonych raportów wybiera tylko te, które dotyczą „Maszyny B1” oraz „Maszyny B4”, a następnie zlicza czasy tylko określonych rodzajów przestojów. Nieplanowany przestój „Maszyny B2” lub „Maszyny B3” spowoduje jedynie spowolnienie pracy całej linii produkcyjnej, dlatego nie uwzględniono ich przy obliczaniu nieplanowanego czasu przestoju.

[4] Dotyczy linii produkcyjnej B

Taka implementacja metody obliczającej nieplanowane przestoje może się okazać niewłaściwa dla innych linii produkcyjnych dysponujących odmiennym rozkładem maszyn. Dlatego tworząc nowy plugin ważne jest, aby w początkowej fazie ich projektowania uzyskać jak najdokładniejsze rozeznanie dotyczące zależności zachodzących pomiędzy poszczególnymi elementami linii.

5.4. Prezentacja danych

W efekcie działania plugina wygenerowana zostaje kolekcja, zawierająca obiekty klasy *DataAnalysis* będące reprezentacją danych wyjściowych. W dalszym etapie pracy systemu lista ta przekazana zostaje do obiektu klasy *DailyReportChart*, którego zadaniem jest wygenerowanie wykresu graficznego reprezentującego dane zawartych w kolekcji. Utworzenie obiektu umożliwiającego obliczenie współczynnika OEE dla wybranej linii produkcyjnej, a następnie przekazanie wyników tej analizy do obiektu generującego wykres graficzny przedstawiono na listingu 25.

Listing 25 Przykładowy sposób implementacji, umożliwiający dokonanie analizy przez wybrany plugin oraz przekazanie jej wyników do obiektu generującego wykres graficzny

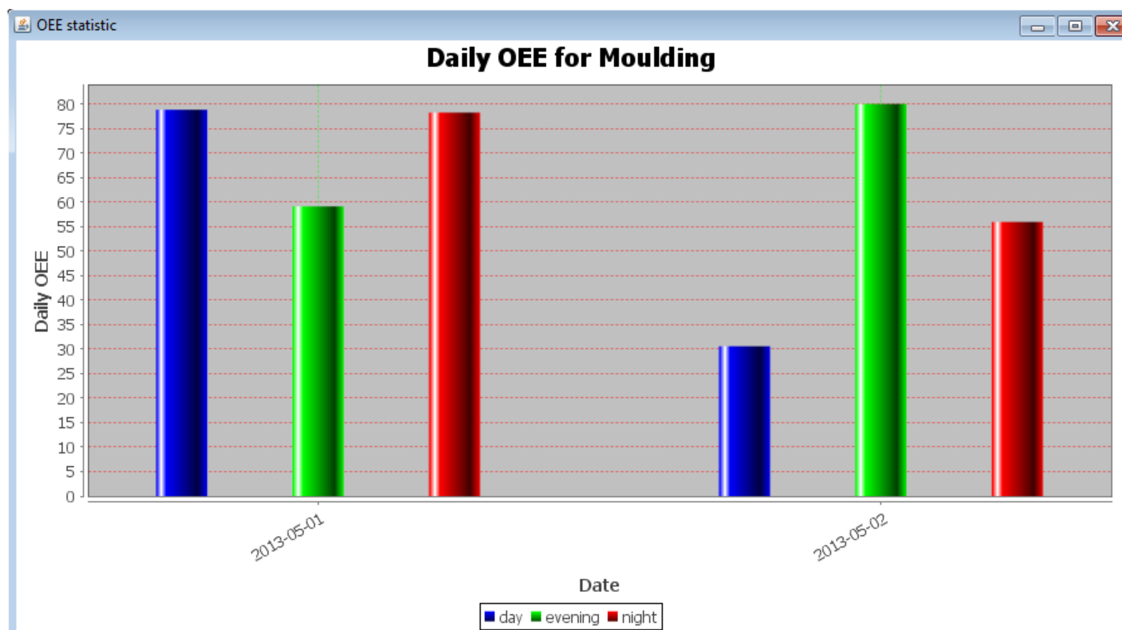
```
...
// utworzenie obiektu analizującego zależności na podstawie przekazanej listy
// raportów
MouldingDailyOEECounter counter =
                         new MouldingDailyOEECounter(reportList, reportType);
// utworzenie obiektu umożliwiającego wygenerowanie wykresu graficznego
new DailyReportChart(desktop, counter.getDataList());
...
```

Definicja klasy *DailyReportChart* stanowi jeden ze sposobów implementacji zewnętrznej biblioteki JFreeChart. W zaprojektowanym prototypie systemu klasa ta pozwala ona prezentowanie danych w formie wykresów słupkowych. Sposób implementacji takiej klasy zaprezentowano na listingu 26.

Listing 26 Fragment kodu implementującego klasę DailyReportChart.

```
public class DailyReportChart {
  ...
  private CategoryDataset createDataset() {
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
     for(DataAnalysis data : dataList){
      dataset.addValue(data.getValue(),data.getSeries(),data.getCategory());
     }
    return data;
  }
  private JFreeChart createChart(CategoryDataset data) {
    JFreeChart chart = ChartFactory.createBarChart(
                         dataList.get(0).getRaportType()+
                         " for "+dataList.get(0).getProductionLineName(),
                         "Date",
                         dataList.get(0).getRaportType(),
                         data,
                         PlotOrientation.VERTICAL,true,true,false);
      ...
      return chart;
     }
  ...
}
```

Metoda createDatset umożliwia stworzenie obiektu klasy *CategoryDataset* stanowiącej zestaw danych akceptowalny przez zewnętrzną bibliotekę JFreeChart. Następnie tak przygotowany zestaw danych przekazywany jest do obiektu klasy *JFreeChart*, który odpowiedzialny jest za przetworzenie ich na postać graficzną. Efektem tych działań jest wykres w postaci zaprezentowanej na rysunku 33.



Rysunek 33 Analiza współczynnika OEE - wykres słupkowy. Źródło: Opracowanie własne

Proponowana implementacja systemu przewiduje generowanie wykresów słupkowych, jednakże nic nie stoi na przeszkodzie, aby rozbudować jej funkcjonalność o możliwość prezentowania danych w formie innego rodzaju wykresów. W tym celu należałoby zaimplementować dodatkowe klasy, analogicznie do klasy *DailyReportChart*, które umożliwiałby realizacje takiej funkcjonalności.

6. Podsumowanie

Analizowanie efektywności produkcyjnej pojedyńczego urządzenia nie stanowi problemu. Istnieje obecnie wiele systemów realizujących taką funkcjonalność. Należy jednak pamiętać, że proces produkcyjny wykonywany może być również przez zespół połączonych ze sobą oraz wzajemnie na siebie oddziałujących maszyn. Przeprowadzenie odpowiedniej analizy w takim przypadku może okazać się bardzo skomplikowane.

Stworzony na potrzeby pracy prototyp przedstawia jeden z możliwych sposobów implementacji generycznego systemu umożliwiającego analizę efektywności linii produkcyjnej. Dużą jego zaletą jest możliwość przystosowania go do współpracy i analizy z każdym rodzajem linii produkcyjnej. Dzięki zastosowaniu systemu pluginów, umożliwiono dostarczenie dokładnie zdefiniowanych relacji zachodzących pomiędzy poszczególnymi urządzeniami. W połączeniu z gromadzonymi w bazie, danymi produkcyjnymi możliwe jest dokładne przeprowadzenie analizy efektywności produkcyjnej takiej linii. Istniejące na rynku systemy, umożliwiające dokonanie podobnej analizy wymagają, zastosowania szeregu urządzeń monitorujących. Proponowane w prototypie rozwiązanie, redukuje koszty związane z zakupem, instalacją oraz konserwacją takich urządzeń, ograniczając je tylko do kosztów związanych z implementacją poszczególnych pluginów.

Wadą zaprojektowanego prototypu jest przede wszystkim uzależnienie pracy systemu od czynnika ludzkiego. Uzyskanie wiarygodnych rezultatów obliczeń, silnie zależy od danych produkcyjnych dostarczanych w postaci dziennych raportów. Oznacza to, że prawidłowe i systematyczne ich tworzenie staje się kluczowym elementem wpływającym na wartości prowadzonych obliczeń współczynnika OEE. Ponadto w celu zdefiniowania nowych wtyczek wymagane jest posiadanie wiedzy programistycznej. Mimo tego, zaimplementowanie przez specjalistę nowego plugina wydaje się dużo mniej kosztownym przedsięwzięciem w stosunku do konieczności stworzenia całego systemu przeznaczonego tylko dla określonej linii produkcyjnej lub wyposarzenia takiej linii w szereg urządzeń monitorujących .

W stworzonym prototypie zaimplementowano tylko podstawowe funkcjonalności umożliwiające gromadzenie i analizowanie danych, na podstawie których dokonywane są wyliczenia współczynnika OEE. Przewidywany kierunek rozwoju zaproponowanego rozwiązania przewiduje wyposażenie go w funkcje umożliwiające:

- dokonywania analizy parametrów wpływających na dostępność, jakość oraz wydajność określonej linii produkcyjnej np.: ilości wadliwych produktów, długości przestoju linii czy rodzaju występujących przestojów,
- generowanie dodatkowych typów wykresów na podstawie przeprowadzanych analiz np.: wykresy kołowe, wykresy punktowe, wykresy liniowe,
- definiowanie nowych linii produkcyjnych przy użyciu kreatora graficznego,
- powiadamianie o nieplanowanych przestojach obsługi technicznej.

Zaprezentowany sposób implementacji z wykorzystaniem pluginów stanowi bazę wyjściową dla stworzonego systemu, a zdefiniowanie nowych funkcjonalności z pewnością pozwoli zwiększyć jej użyteczności.

7. Bibliografia

[1]. **Mazurek Wojciech.** Wskaźnik OEE - Teoria i praktyka. [Online] http://www.oee.pl/oee.pdf [data dostępu: 17.11.2012].

[2]. Dokumentacja Golem OEE. [Online] http://www.neuron.com.pl/golemoee.html [data dostępu: 14.11.2012].

[3]. Dokumentacja Computerised Maintenance Management System. [Online] http://www.essltd.ie/systems-page50170.html [data dostępu: 20.11.2012].

[4]. Dokumentacja Provideam OEE. [Online] http://www.provideam.com/ [data dostępu: 12.12.2012].

[5]. Dokumentcaja System Monitorowania Linii Produkcyjnych (SMLP). [Online] http://progresja.com.pl/doradztwo/doradztwo-it/150 [data dostępu: 06.12.2012].

[6]. **Holzner Steve.** *Eclipse.* : Helion, 2004.

[7]. **Barteczko Krzysztof.** Programowanie obiektowe w języku Java. [Online] http://edu.pjwstk.edu.pl/wyklady/poj/scb/index.html [data dostępu: 23.04.2013].

[8]. Java SE 7 Swing APIs and Developer Guides. [Online] http://docs.oracle.com/javase/7/docs/technotes/guides/swing/ [data dostępu: 05.07.2013].

[9]. MiG Lauout Java Layout for Swing and SWT. [Online] http://www.miglayout.com/ [data dostępu: 18.07.2013].

[10]. JFreeChart. [Online] http://www.jfree.org/jfreechart/ [data dostępu: 11.03.2013]

[11]. [Online] http://www.eclipse.org [data dostępu: 14.03.2013].

[12]. JFreeChart - Dokumentacja API. [Online] http://www.jfree.org/jfreechart/api/javadoc/index.html [data dostępu: 11.03.2013].

[13]. Java Simple Plugin Framework. [Online] https://code.google.com/p/jspf/ [data dostępu: 04.08.2013].

[14]. Hibernate - JBoss Community. [Online] http://www.hibernate.org/ [data dostępu: 4.08.2013].

[15]. [Online] http://dev.mysql.com/doc/refman/5.5/en/introduction.html. [data dostępu: 25.07.2013].

[16]. Zarządzanie produkcją, wydajność i planowanie produkcji - VIX. [Online] http://www.vix.com.pl/wskaznik-oee--overall-equipment-effectiveness-,c/ [data dostępu: 02.01.2013].

[17]. **Dubois Paul.** *MySQL Language Reference.* 2013.

[18]. **Bloch Joshua.** *Java. Efektywne programowanie. Wydanie II.* brak miejsca : Helion, 2009.

[19]. **Horstmann Cay S i Cornell Gary.** *Core Java Volume I--Fundamentals (9th Edition).* 2012.

8. Spis rysunków

Rysunek 1 Graficzna reprezentacja wskaźnika OEE 4
Rysunek 2 Przykład organizacji linii produkcyjnych 5
Rysunek 3 Graficzna prezentacja wskaźnika OEE dla określonego urządzenia – GOLEM OEE SuperVisor 9
Rysunek 4 Raport efektywności dla określonego zlecenia - GOLEM OEE SuperVisor. 9
Rysunek 5 Wykresy wskaźnika OEE na podstawie danych napływających w czasie rzeczywistym – CMMS firmy ESS Ltd. 10
Rysunek 6 Wykresy wskaźnika OEE na podstawie danych historycznych – CMMS firmy ESS Ltd. 11
Rysunek 7 Dane szczegółowe linii produkcyjnej – CMMS firmy ESS Ltd. 11
Rysunek 8 Symulacja pracy system w czasie rzeczywistym – Provideam OEE. 13
Rysunek 9 Konfigurowanie źródła danych dla określonego urządzenia – Provideam OEE. 13
Rysunek 10 Interfejs użytkownika umożliwiający dodawania danych produkcyjnych – Provideam OEE 14
Rysunek 11 Wykresy wskaźnika OEE oraz produkcji dla wybranego urządzenia – Provideam OEE 15
Rysunek 12 Główny panel użytkowania – SMLP. 16
Rysunek 13 Raport dotyczący wskaźnika OEE – SMLP. 16
Rysunek 14 Schemat współpracy systemu głównego z pluginami. 19
Rysunek 15 Przykład dziedziczenia w Javie. 23
Rysunek 16 Proces uruchamiania programu w języku Java. 24
Rysunek 17 Hierarchia komponentów biblioteki SWING. 25
Rysunek 18 Przykładowe rodzaje rozkładu komponentów. 26
Rysunek 19 Przykładowa siatka rozkładu - biblioteka MigLayout 27
Rysunek 20 Przykładowe umiejscowienie komponentu w siatce rozkładu – MigLayout. 28
Rysunek 21 Wykres kołowy 3D - biblioteka JFreeChart. 30
Rysunek 22 Przykład dostępnych typów wykresów – JFreeChart 31
Rysunek 23 Eclipse Marketplace. 32
Rysunek 24 Tworzenie nowego projektu - Eclipse Juno. 33
Rysunek 25 Przykładowa struktura katalogowa dla projektu Java – Eclipse Juno. 34
Rysunek 26 Wskazanie potencjalnego błędu w implementacji - Eclipse Juno. 34
Rysunek 27 Asystent wprowadzenia - Eclipse Juno. 35
Rysunek 28 Ogólna architektura Hibernate. 38
Rysunek 29 Diagram UML reprezentujący GUI systemu. 43
Rysunek 30 Schemat relacyjnej bazy danych systemu. 49
Rysunek 31 Diagram UML pluginu dla przykładowej linii produkcyjnej 50
Rysunek 32 Formularz do przeprowadzania analizy OEE. 51
Rysunek 33 Analiza współczynnika OEE - wykres słupkowy. 54

9. Spis tabel

Tabela 1 Wybrane typy danych w systemie mySQL. Źródło: Opracowanie własne...... **Błąd! Nie zdefiniowano zakładki.**
Tabela 2 Wybrane typy adnotacji - Hibernate Validator. Źródło: Opracowanie własne **Błąd! Nie zdefiniowano zakładki.**

10. Spis listingów

Listing 1 Przykład implementacji MigLayout w odniesieniu do wybranego komponentu Swing........ 26
Listing 2 Przykładowe zastosowania ograniczeń biblioteki MigLayout.. 27
Listing 3 Główne metody umożliwiające dostęp do istniejących pluginów. 28
Listing 4 Sposób uzyskiwania kolekcji pluginów.. 29
Listing 5 Przykład zastosowania adnotacji @PluginImplementation. 29
Listing 6 Przykład zastosowania adnotacji @InjectPlugin. 29
Listing 7 Przykład zastosowania adnotacji @PluginLoaded. 29
Listing 8 Przykład zastosowania adnotacji @Init. 30
Listing 9 Przykład zastosowania adnotacji @Timer.. 30
Listing 10 Tworzenie zestawu danych dla wykresu kołowego - biblioteka JFreeChart. 31
Listing 11 Tworzenie obiektu reprezentującego wykres kołowy – JfreeChart. 31
Listing 12 Przykład mapowania obiektów zdefiniowana w pliku XML.. 38
Listing 13.Sposób mapowania obiektów za pomocą adnotacji.. 38
Listing 14 Przykład klasy spełniającej reguły POJO 40
Listing 15Sposób implementacji metody fabrykującej getRole.. 43
Listing 16 Zestaw zdefiniowanych własności pliku konfiguracyjnego hibernate.cfg.xml 44
Listing 17 Przykładowa implementacja klasy POJO z wykorzystaniem adnotacji.............................. 44
Listing 18 Przykład implementacji relacji 1:1... 45
Listing 19 Przykład implementacji relacji 1:N oraz N:1 46
Listing 20 Sposób implementacji relacji N:M.. 47
Listing 21 Przykład wykorzystania interfejsów biblioteki Hibernate 47
Listing 22 Przykład implementacji metody getSelectedUsers 48
Listing 23 Sposób implementacji metody findDailyReports 51
Listing 24 Przykładowy sposób implementacji metody calculateUnplannedDownTime.................... 52
Listing 25 Przykładowy sposób implementacji, umożliwiający dokonanie analizy przez wybrany plugin oraz przekazanie jej wyników do obiektu generującego wykres graficzny 53
Listing 26 Fragment kodu implementującego klasę DailyReportChart... 53