Przedmiot: Metody Inżynierii Wiedzy – Projekt
Tytuł: SoftEng_MVC

Monika Mroczek mailto:m_mroczek@poczta.fm
AiR, IV rok, lato 2007

24 kwietnia 2007

Spis treści

1 Sformułowanie problematyki i celu projektu **3**

2 Model View Controller - wprowadzenie **3**
2.1 Definicja wzorca projektowego 3
2.2 Koncepcja MVC 3
2.2.1 Model 4
2.2.2 Widok 5
2.2.3 Kontroler 5
2.2.4 Wzajemne relacje między komponentami 6

3 Wykorzystywane w MVC wzorce projektowe **6**
3.1 Obserwator 6
3.2 Kompozyt 8
3.3 Strategia 8
3.4 Katalogi wzorców projektowych 9

4 Zastosowanie MVC **9**
4.1 Smalltalk-80 9
4.2 Ruby 10
4.2.1 Ruby on Rails 10
4.3 Python 12
4.3.1 Django 12
4.3.2 Pylons 12
4.4 PHP 13
4.4.1 CakePHP 13
4.4.2 Symfony 14
4.5 .NET 16
4.5.1 ASP.NET 16
4.6 Java 16
4.6.1 Swing 16
4.6.2 J2EE 17

5 MVC w przykładach **22**
5.1 Przykłady - ogólna koncepcja 22
5.1.1 Aplikacja GUI 22
5.1.2 Aplikacja webowa 23
5.2 Przykłady - konkretne frameworki 28
5.2.1 Struts 28
5.2.2 ASP.NET 30
5.2.3 GEF i EMF 36

6 Podsumowanie **39**

1. Sformułowanie problematyki i celu projektu

Celem projektu realizowanego z przedmiotu Metody Inżynierii Wiedzy będzie omówienie koncepcji architektury typu *MVC Model View Controller* oraz sposobu jej realizacji w różnych technikach programowania wraz z prezentacją aplikacji zaprojektowanych z wykorzystaniem wzorca *MVC*.

Praca została podzielona na dwie części. Pierwsza część ma charakter teoretyczny. Podana zostanie definicja wzorca projektowego, idea wzorca *Model View Controller* oraz krótki opis wzorców projektowych używanych w *MVC*. W drugiej części pracy zostanie odtworzona historia *MVC* począwszy od Smalltalk po Javę. Zostanie ona poparta przykładami aplikacji, głównie w postaci diagramów behawioralnych UML.

2. Model View Controller - wprowadzenie

2.1. Definicja wzorca projektowego

MVC Model View Controller jest zorientowanym obiektowo wzorcem projektowym. Wzorce projektowe stosuje się w celu zapewnienia dobrych, sprawdzonych rozwiązań dla często występujących problemów. Katalogują one te sposoby interakcji między obiektami, które będą najbardziej przydatne w przyszłych projektach. Dzięki wzorcom uzyskuje się lepszą jakościowo dokumentację oraz szybszy przebieg prac projektowych dzięki zastosowaniu wcześniej sprawdzonych technik.

Poniżej podano definicję wzorca projektowego pochodzącą z [1]
"Wzorzec identyfikuje i specyfikuje pewną abstrakcję, której poziom znajduje się powyżej poziomu abstrakcji pojedynczej klasy, instancji lub komponentu. Wzorce projektowe to opisy komunikujących się ze sobą obiektów i klas, które przerabia się w celu rozwiązania ogólnego problemu projektowego w danym kontekście. Wzorzec projektowy wskazuje, wyodrębnia i identyfikuje najważniejsze aspekty typowego rozwiązania, dzięki którym staje się ono przydatne przy tworzeniu dających się ponownie wykorzystać projektów obiektowych. Określa uczestniczące w nim klasy i ich egzemplarze, ich rolę i współpracę oraz podział zobowiązań. Każdy wzorzec projektowy dotyczy konkretnego problemu lub zagadnienia projektowania obiektowego. Opisuje, kiedy można go zastosować, czy można go użyć, biorąc pod uwagę inne ograniczenia projektowe oraz konsekwencje, wady i zalety jego zastosowania."

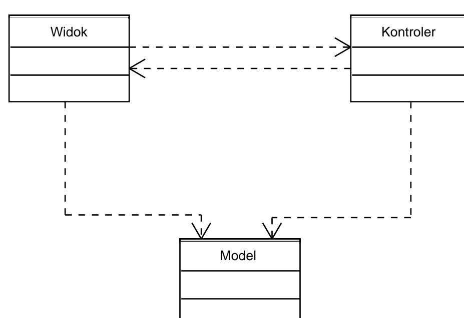
2.2. Koncepcja MVC

Wzorzec *Model View Controller* powstał w późnych latach 70. w laboratoriach PARC (Palo Alto Research Centre) firmy Xerox jako szkielet opracowany przez Trygve Reenskauga dla języka Smalltalk. Architektura MVC była początkowo wykorzystywana w implementacjach graficznego interfejsu użytkownika. Następnie została ona zaadaptowana dla aplikacji WWW, ale z pewnymi modyfikacjami, np. obsługą nawigacji między stronami jako jednym z zadań kontrolera.

Architektura *Model View Controller* zakłada wyodrębnienie trzech podstawowych komponentów aplikacji:

- modelu danych, reprezentującego pewne informacje z modelu dziedziny – *Model*,

- widoku, będącego ekranową reprezentacją modelu – *View*,
- kontrolera, implementującego logikę sterowania – *Controller*.



Rysunek 1: Wzorzec MVC

Przed *MVC* te trzy komponenty były na ogół łączone. *MVC*, patrz Rys. 1, rozdziela je, aby zwiększyć elastyczność oraz możliwość wielokrotnego wykorzystania. Rozdzielenie to skutkuje również tym, że modyfikacje jednego komponentu minimalnie wpływają na pozostałe. Kontroler analizuje żądanie, model zajmuje się przetwarzaniem danych, widok zaś zajmuje się przedstawieniem ich użytkownikowi.

2.2.1. Model

Model nie posiada żadnej reprezentacji wizualnej. Zawiera dane oraz udostępnia operacje niezwiązane z obsługą interfejsu użytkownika, tzw. funkcje logiki biznesowej. Model jest niezależny od widoku.
Funkcje:

- dostarcza rdzenia funkcjonalności aplikacji,
- rejestruje zależne widoki i kontrolery i zawiadamia zależne komponenty o zmianie danych.

Model Pasywny W przypadku pasywnym model jest całkowicie odseparowany. Kontroler informuje widok, gdy wykonywane na modelu operacje wymagają uaktualnienia widoku. Mechanizm ten jest wykorzystywany w aplikacjach webowych, w których nie ma trwałej sesji i z każdym odwołaniem interfejs jest budowany od nowa.

Model Aktywny W aktywnym modelu, klasy modelu posiadają mechanizm zawiadomień. Gdy tylko dane modelu się zmieniają, model powiadamia wszystkie zależące od niego widoki, by te mogły się uaktualnić. Jest to przykład rozdzielenie obiektów, tak by zmiany w jednym z nich miały wpływ na inne obiekty, bez konieczności zmieniania ich wewnętrznej struktury. W tym celu wykorzystuje się wzorzec *Obserwator*, opisany w Rozdziale 3.1. Model pełni rolę Obserwowanego, natomiast widok działa jak Obserwator modelu. Mechanizm ten jest wykorzystywany w aplikacjach typu GUI.

Model Prezentacji i Model Dziedziny Zazwyczaj model jest obiektem należącym do wzorca *Domain Model* (Modelu Dziedziny), który przypomina model bazy danych. Implementuje on zarówno dane jak i operacje na nich. Natomiast stan i zachowanie widoku przechowywane jest w *Presentation Model* (Model Prezentacji) lub *Application Model* (Model Aplikacji). Widok albo przechowuje cały swój stan w Modelu Prezentacji albo go z nim synchronizuje. Model Prezentacji stanowi warstwę pośrednią pomiędzy Modelem Dziedziny a widokiem oraz dostarcza odpowiednich interfejsów w celu zminimalizowania decyzji podejmowanych przed widok. Przykładowo Model Prezentacji wykorzystywany jest w celu zmiany sposobu wyświetlania widgetów, np. zmiany ich koloru. Funkcjonalność ta jest oczywiście niezależna od widoku, natomiast nie należy też do Dziedziny Modelu. Model Prezentacji stanowi tu nowy rodzaj obiektu, niezależny od widgetów, ale dotyczący tylko i wyłącznie warstwy prezentacji.

2.2.2. Widok

Widok służy do prezentowania danych i reprezentuje wyjście aplikacji. Najczęściej używanym formatem wyjściowym jest HTML, ale wyniki mogą być prezentowane w postaci XML, plików graficznych, plików PDF a także w wielu innych formatach. Widok porozumiewa się z modelem w celu odczytania danych, a następnie wyświetlenia ich na ekranie użytkownika. Widok zazwyczaj ma wolny dostęp do modelu, ale nie powinien on ingerować w jego stan.
Funkcje:

- tworzy i inicjalizuje swój kontroler,
- wyświetla informacje dla użytkownika,
- pobiera dane z modelu,
- implementuje operacje aktualizacji.

Widok może być implementowany w postaci *Template View* (Szablon Widoku). Idea tego wzorca polega na umieszczeniu odpowiednich znaczników w kodzie gotowej strony HTML, które podczas przetwarzania strony można zamienić na wywołania dynamicznie pobierające odpowiednie informacje. Wzorzec *Template View* jest wykorzystywany w ASP, JSP czy JSF.

Cechą *MVC* jest to, iż widoki mogą być zagnieżdżone. Idąc za przykładem opisanym w [1] panel sterowania może być złożonym widokiem zawierającym zagnieżdżone widoki przycisków. Tego typu ogólniejsze rozwiązanie opisane jest przez wzorzec *Kompozyt*, patrz Rozdział 3.2. Umożliwia on tworzenie hierarchii klas, w których pewne podklasy definiują obiekty pierwotne (np. przyciski), a inne obiekty złożone (np. panel sterowania) zestawiające obiekty pierwotne.

2.2.3. Kontroler

Kontroler jest odpowiedzialny za sposób, w jaki interfejs użytkownika odpowiada na operacje przez niego wykonywane.
Funkcje:

- pobiera wejście od użytkownika w postaci zdarzeń,
- reaguje na akcje użytkownika odwzorowując je na akcje zawarte w modelu oraz na zmiany w widoku.

W tradycyjnym *MVC* kontroler nie jest mediatorem pomiędzy widokiem i modelem, innymi słowy kontroler nie znajduje się pomiędzy widokiem i modelem. Zarówno kontroler jak i widok mają jednakowy dostęp do modelu. Kontroler nie kopiuje danych z modelu do widoku, pomimo że mógłby pobierać dane od modelu i oznajmiać widokowi, iż zaszły zmiany w modelu.

Kontroler często bywa interpretowany w różny sposób. Często jest uważany za element reprezentujący działanie aplikacji przy założeniu, że model reprezentuje dane statyczne. Kontroler jest równocześnie odpowiedzialny za przetwarzanie żądań użytkownika oraz logikę aplikacji. Tak rozumiany kontroler nie pozwala na zmianę warstwy prezentacji bez modyfikacji logiki. Z tego względu kontroler nie powinien zawierać logiki aplikacji a jedynie odwołanie do niej.

MVC rozdziela widok od kontrolera. Przydatne staje się to wówczas, gdy trzeba zaimplementować w widoku różne strategie odpowiadania na akcje użytkownika bez zmiany jego wizualnej reprezentacji. Przykładowo można zmienić sposób, w jaki widok reaguje na funkcje klawiatury bądź sprawić, by widok był nieaktywny i nie reagował na akcje użytkownika. Wystarczy zmienić kontroler. Obecnie rozwiązanie to przestaje być aktualne, ponieważ w praktyce każdy widok posiada jeden kontroler i z tego powodu rozdzielanie tych komponentów nie jest realizowane. W większości GUI *MVC* widok i kontroler są powiązane w jeden obiekt (ang. *Document View*).

Rola kontrolera w przypadku aplikacji internetowych jest odmienna, gdzie pełni on kluczową rolę. Wszystkie żądania są wysyłane do kontrolera, który odwzorowuje żądania w wywołania metod modelu. Następnie wyniki tych działań przekazywane są do widoku. Do zadań kontrolera należy również dbanie o bezpieczeństwo aplikacji oraz walidacja danych. W aplikacji webowej kontroler jest kombinacją wzorców *Front Controller*, który obsługuje wszystkie żądania przesyłane do witryny kierując je do jednego obiektu oraz *Application Controller*, który kontroluje całą aplikację: określa, jaka logikę dziedziny należy wykonać oraz jaki widok wyświetlić w odpowiedzi.

Związek widok - kontroler to przykład wzorca projektowego *Strategia*, patrz Rozdział 3.3. Strategia reprezentuje algorytm.

2.2.4. Wzajemne relacje między komponentami

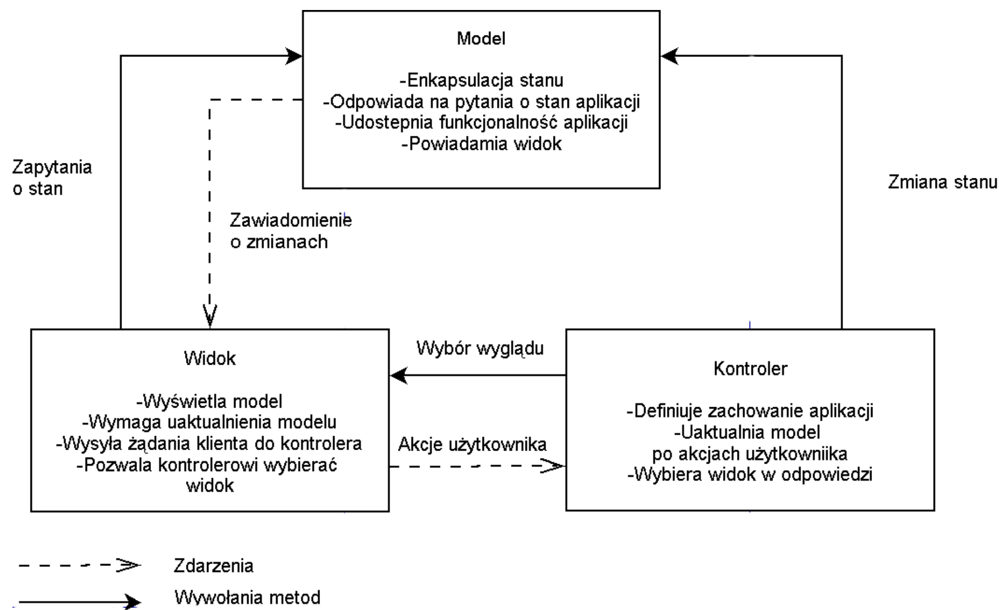
Na Rys. 2 przedstawiono relacje pomiędzy komponentami należącymi do *MVC*.

3. Wykorzystywane w MVC wzorce projektowe

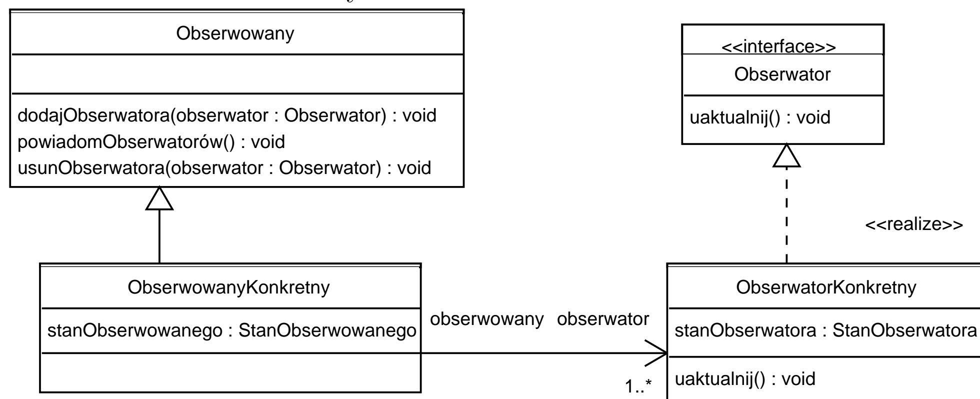
Główne związki w architekturze MVC są zadane przez wzorce projektowe *Obserwator*, *Kompozyt* i *Strategia*. Mniejszą rolę pełnią wzorce *Dekorator* (do przewijania widoku), czy *Metoda Wytwórcza* do wyspecyfikowania domyślnej klasy będącej kontrolerem.
Opisy wzorców projektowych zostały podane za [1].

3.1. Obserwator

Wzorzec *Obserwator*, przedstawiony na Rys. 3, określa zależności jeden do wielu między obiektami. Gdy jeden obiekt zmienia stan, obiekty od niego zależne są o tym automatycznie powiadamiane i uaktualniane. Uczestnikami są:



Rysunek 2: Wzorzec MVC

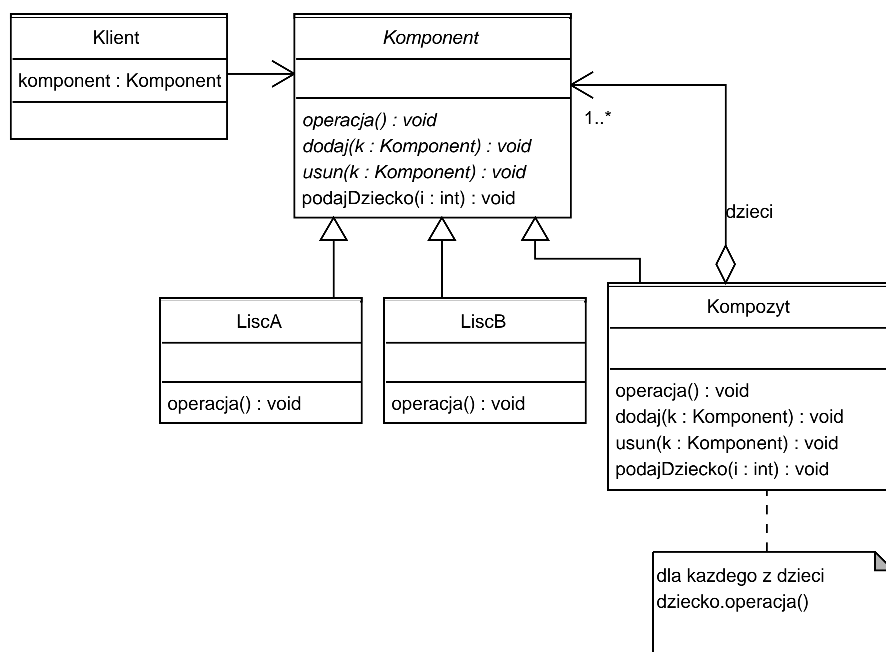


Rysunek 3: Wzorzec Obserwator

- Obserwowany - zapewnia interfejs dołączania i odłączania Obserwatorów, zna swoich Obserwatorów.
- Obserwator - definiuje interfejs uaktualniania.
- Obserwowany Konkretny - przechowuje stan i wysyła powiadomienie do swoich Obserwatorów.
- Obserwator Konkretny - implementuje interfejs Obserwatora do uaktualniania oraz utrzymuje odwołanie do obiektu Obserwatowany Konkretny.

Obserwowany zna swoich Obserwatorów. Przechowuje on stan, który interesuje Obserwatorów. W przypadku zmiany stanu informuje o tym wszystkich swoich Obserwatorów. Obserwator posiada odwołanie do Obserwowanego. Po otrzymaniu powiadomienia o zmianie stanu wysyła zapytanie dotyczące tej zmiany oraz uaktualnia swój stan zgodnie ze stanem Obserwatora.

3.2. Kompozyt



Rysunek 4: Wzorzec Kompozyt

Wzorzec *Kompozyt*, przedstawiony na Rys. 4, pozwala tworzyć drzewiaste struktury reprezentujące hierarchie typu część - całość. Uczestnikami są:

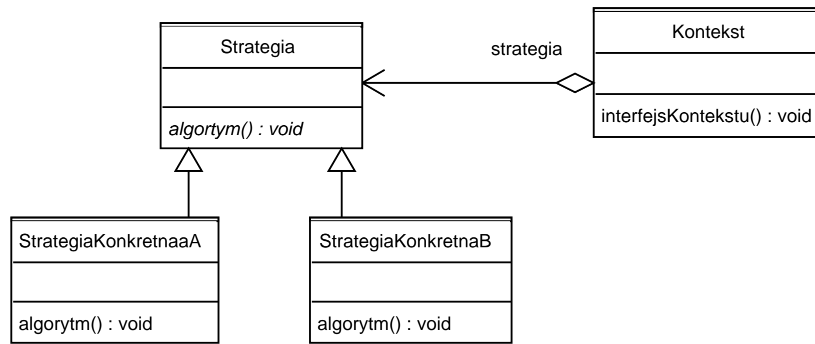
- Komponent - definiuje interfejs umożliwiający dostęp i zarządzanie komponentami-dziećmi, deklaruje interfejs składanych obiektów oraz implementuje domyślne zachowanie.
- Liść reprezentuje obiekty pierwotne i definiuje ich zachowanie.
- Kompozyt przechowuje komponenty będące dziećmi oraz implementuje operacje z interfejsu Komponentu.
- Klient manipuluje obiektami.

Klienci używają interfejsu z klasy Komponent do komunikowania się z obiektami struktury. Jeśli odbiorcą jest Liść żądania wykonywane są bezpośrednio. Natomiast jeśli odbiorcą jest Kompozyt, to żądania przekazywane są jego dzieciom.

3.3. Strategia

Wzorzec *Strategia*, przedstawiony na Rys. 5, definiuje rodzinę algorytmów. Umożliwia zmienianie algorytmów niezależnie od klientów, którzy ich używają. W metodzie klasy abstrakcyjnej można umieścić wspólne części algorytmu, w podklasach ich specjalizację. Uczestnicy:

- Strategia deklaruje interfejs wspólny dla wszystkich obsługiwanych algorytmów.
- StrategiaKonkretna implementuje algorytm wykorzystując interfejs z klasy Strategia.



Rysunek 5: Wzorzec Strategia

- Kontekst posiada odwołanie do obiektu Strategia, jest konfigurowany za pomocą obiektu Strategia Konkretna. Przekazuje żądania korzystających z niego obiektów klasie Strategia.

Kontekst decyduje o tym, która ze Strategii Konkretnych zostanie wykonana. Decyzja jest podejmowana na podstawie żądania programu klienta.

3.4. Katalogi wzorców projektowych

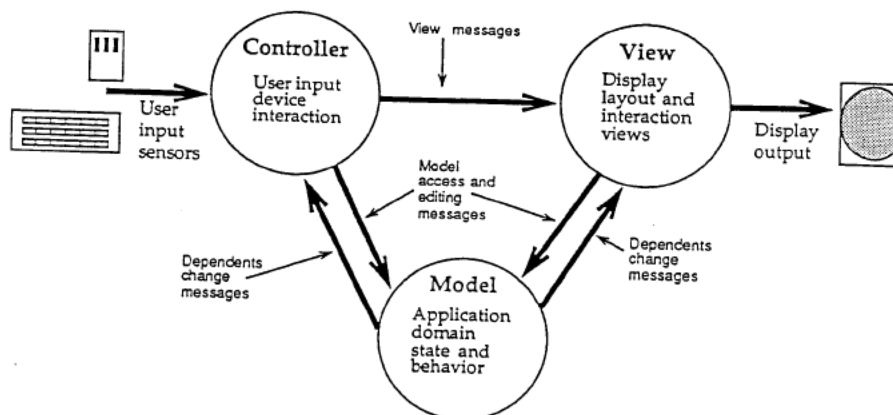
Informacje o wzorcach projektowych zostały zaczerpnięte z następująych źródeł:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe: elementy oprogramowania obiektowego wielokrotnego użytku*, patrz [1].
- Martin Fowler, David Rice, Matthiew Foemmel, Edward Hieatt, Robert Mee, Rendy Stafford, *Architektura systemów zarządzania przedsiębiorstwem Wzorce projektowe*, patrz [3].
- *Development of Further Patterns of Enterprise Application Architecture*, patrz [6].
- *Core J2EE Pattern Catalog*, patrz [7].
- *Microsoft patterns & practices*, patrz [8].
- *Design Patterns*, patrz [9].
- *Patterns Catalog*, patrz [10].

4. Zastosowanie MVC

4.1. Smalltalk-80

Implementacja architektury *MVC* w Smalltalk-80, patrz Rys. 6, składa się z trzech klas abstrakcyjnych: `Model`, `View` oraz `Controller` oraz klas potomnych definiujących specyficzne zachowanie każdego z komponentów triady *MVC*.



Rysunek 6: MVC w Smalltalk-80

Klasa `Model` implementuje dziedzinę problemu, reprezentowaną przez instancje klasy `Object`. Z każdym modelem jest związanych jeden lub więcej widoków oraz kontrolerów. W celu powiadomienia ich o zmianie stanu, model posiada odwołanie do listy zarejestrowanych kontrolerów i widoków, tzw. `DependentsCollection`.

Klasa `View` posiada odwołanie do dokładnie jednego modelu oraz jednego kontrolera. Widok pobiera dane z modelu oraz wyświetla je na ekranie. Widoki mogą mieć zero lub więcej widoków podrzędnych. Każdy widok posiada odwołanie do swoich widoków podrzędnych oraz nadrzędnych, tworząc strukturę drzewa. Za zachowanie głównego okna (np. rozmiar) odpowiedzialna jest klasa `StandardSystemView`. Istnieje szereg klas potomnych obsługujących specyficzne typy prezentacji: `ListView` reprezentujące menu, `SelectionInListView` reprezentujące listy wyboru, `Prompter` i `Confirmer` reprezentujące dialogi, `TextView` dostarczające pól tekstowych.

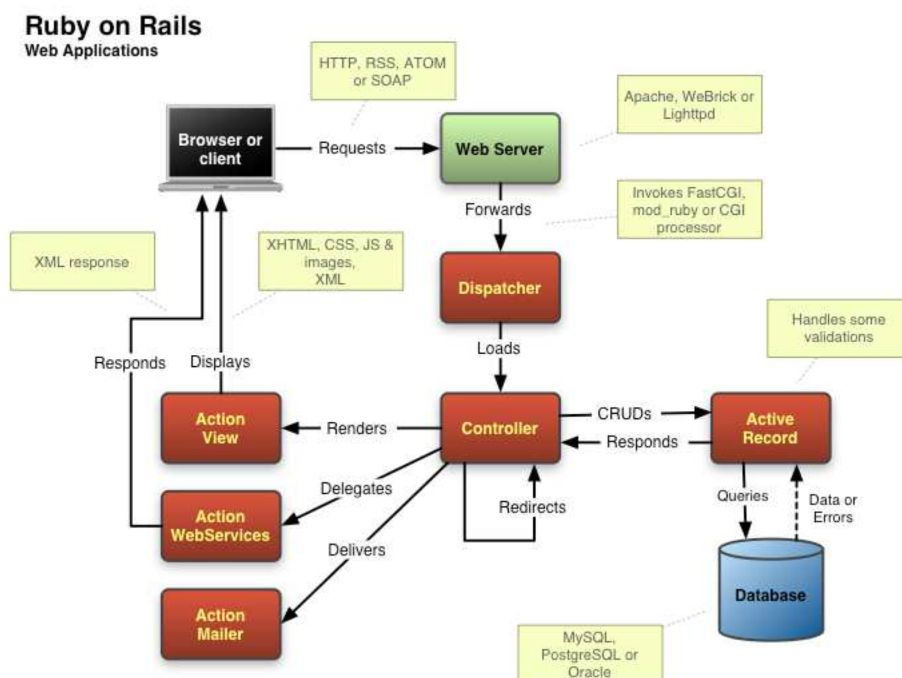
Klasa `Controller` zapewnia komunikację pomiędzy widokiem, modelem a urządzeniami wejścia (będącymi instancjami klasy `InputSensor`). Każdy kontroler posiada odwołanie tylko do jednego widoku. Tylko jeden kontroler może obsługiwać akcje użytkownika w danym momencie. W tym celu główny `ControlManager` przekazuje sterowanie do tego z kontrolerów, którego widok jest aktywny. Następnie sterowanie zostaje przekazane do kontrolera związanego z widokiem podrzędnym. Za podstawowe operacje zamykania, maksymalizacji okna głównego odpowiedzialna jest klasa `StandardSystemController`, `NoController` reprezentuje obsługę widoków tylko do odczytu, klasa `MouseMenuController` odpowiada za obsługę myszy i wywołuje klasę `PopUpMenu` obsługującą pop-up menu, klasa `TextEditor` obsługuje standardową funkcjonalność pól tekstowych, klasa `ParagraphEditor` - pola tekstowe.

4.2. Ruby

4.2.1. Ruby on Rails

Ruby on Rails (często nazywany RoR lub po prostu Rails) to framework open-sourcowy do szybkiego tworzenia aplikacji webowych napisany w języku Ruby. RoR zapewnia pełną implementację wszystkich komponentów *MVC*, patrz Rys. 7. Widoki to zazwyczaj strony XHTL,

natomiast model i kontroler są zaimplementowane jako klasy Ruby. Żądanie generowane przez klienta jest przyjmowane przez kontroler, który generuje odpowiedź w formie widoku wysyłanego do klienta (najczęściej kontroler w celu wygenerowania odpowiedzi pobiera dane z modelu).



Rysunek 7: MVC Ruby on Rails

Modele w Rails są reprezentowane przez klasy dziedziczące po `ActiveRecord`. Ruby zapewnia mechanizm ORM - mapowanie obiektowo-relacyjnego. Obsługiwane bazy to min. MySQL, Microsoft SQL Server, PostgreSQL, Oracle, SQLite, Sybase, DB2, FireBird. Zapewniona została walidacja danych oraz mapowanie relacji między tabelami dla klas modelu. Mimo, że mechanizm modelu w Rails został zaimplementowany z myślą o bazach danych, istnieje możliwość utworzenia klasy, która nie dziedziczy po `ActiveRecord`, mogącej pobierać dane z pliku XML. Klasy modelu zapewniają obsługę operacji *CRUD Create/Retrieve/Update/Delete*.

Widoki są definiowane przez klasę `ActionView`, zawierającą kod HTML oraz Ruby. Jeden widok odpowiada zazwyczaj jednej akcji kontrolera. Strony Rails przypominają strony JSP albo ASP i są zapisywane są w plikach .rhtml. Ponadto widoki mogą być tworzone w formacie XML, wówczas zapisywane będą jako .rxml. Widoki w Rails korzystają z helperów, które ułatwiają tworzenie elementów takich jak odnośniki, obrazki, listy itp. Rails wspiera także szablony.

Kontrolery to klasy dziedziczące po `ActionController`. Po otrzymaniu żądania od klienta, serwer tworzy instancję odpowiedniego kontrolera. Następnie wywoływane są metody odpowiadające danej akcji, a potem wywoływany jest odpowiedni widok. Zmienne instancji są przekazywane do widoku.

Zalety Do zalet Rails należy bogactwo IDE, wbudowane mechanizmy testowania, dobra dokumentacja oraz konwencja ponad konfiguracją (Rails nie wymaga plików konfiguracyjnych).

Ponadto Rails posiada własny ORM oraz mechanizm *scaffolding*, czyli automatycznej generacji: modeli, widoków, kontrolerów oraz migracji.

4.3. Python

4.3.1. Django

Django jest wysokopoziomowym frameworkiem służącym do tworzenia aplikacji internetowych (szczególnie aplikacji CMS) napisanym w Pythonie. Jest on jednym z najpopularniejszych frameworków dla tego języka. Do zalet Django należy szybkość, stabilne biblioteki, wyższy poziom abstrakcji.

Django stosuje wzorzec *MVC*, ale z pewną różnicą. Według przyjętej konwencji nazewnictwa jest to *Model Template View*. *Model* opisuje strukturę tabeli w bazie danych, *Template* to szablon opisujący wygląd, a *View*, czyli widok, pełni rolę kontrolera i steruje działaniem aplikacji. Różnica polega tylko i wyłącznie na innym nazwaniu komponentów.

Model w Django służy do mapowania tabel z bazy danych. Jest on klasą Pythona dziedziczącą po `djang.db.models.Model`. Każdy atrybut modelu określa pole w bazie danych. Operacje na bazach danych są przeprowadzane przez ORM Django (bazy PostgreSQL, MySQL, SQLite, Microsoft SQL Server). Pełna obiektowość uniezależnia kod od typu bazy danych.

Widok (w terminologii Django, a tradycyjnie kontroler) jest metodą modułu Pythona a nie klasy Pythona. Jego zadaniem jest pobranie odpowiednich danych z bazy oraz wyświetlenie na stronie szablonu. Dodatkowo odpowiada za autoryzację oraz walidację danych. Każdy widok to funkcja przyjmująca jeden obowiązkowy parametr `request` i zwracająca obiekt `HttpRequest`. W Django można korzystać z tzw. widoków generycznych wykonujących z góry określone czynności, np. przekierowanie na podany URL

Szablon to plik tekstowy zawierający kod HTML oraz tagi - zmienne, za które zostanie wstawiona określona treść bądź też tagi kontrolujące logikę szablonu. Dodatkowo Django wspiera XML, CSS, JavaScript.

4.3.2. Pylons

Pylons to framework stworzony w Pythonie, służący do szybkiego tworzenia skalowalnych aplikacji internetowych. Wzorowany jest na Ruby on Rails. Pylons opiera się na wzorcu projektowym *MVC*. Struktura frameworka:

- *Paste* - zbiór komponentów umożliwiających łatwe i dynamiczne generowanie aplikacji.
- *Routes* - maper adresów URL z kontrolerami. Umożliwia tworzenie prostych reguł określających jaka akcja ma być podjęta przy pojawieniu się określonego URLa.
- *Myghty* - system szablonów.
- *SQLAlchemy* - opcjonalny ORM, domyślnie Pylons korzysta ze standardowego SQLObject Pythona.

Porównanie Poniżej przedstawione zostaną cechy obydwu frameworków.
Zalety Django:

- lepsza dokumentacja,
- lepsza obsługa formularzy,
- generic views,własny ORM,
- szybkość.

Zalety Pylons:

- elastyczność,
- prostota budowania adresów URL bez konieczności posiadania wiedzy o wyrażeniach regularnych, helpery do budowania adresów URL,
- wbudowana implementacja AJAXa,
- duże możliwości szablonów Myghty.

4.4. PHP

4.4.1. CakePHP

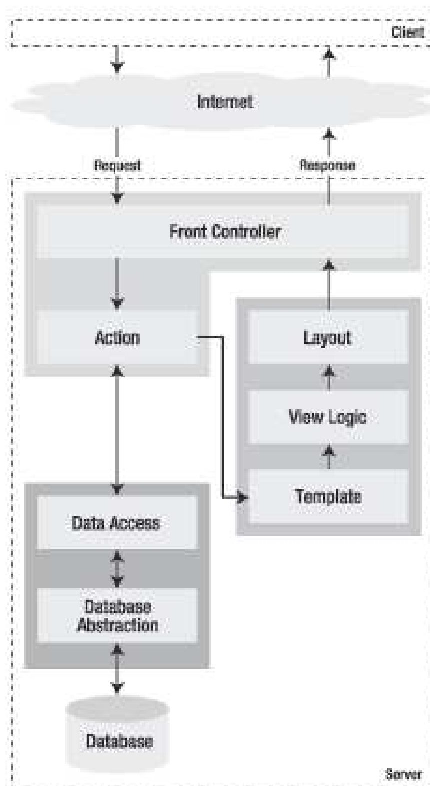
CakePHP jest frameworkiem języka PHP, który powstał na bazie inspiracji Ruby on Rails - patrz Rozdział 4.2.1. CakePHP opiera się na idei wykorzystania konwencji nazewnictwa i miejsca przechowywania elementów aplikacji w celu zminimalizowania konfiguracji. Struktura katalogów aplikacji jest ustalona. Katalog „app" reprezentuje samą aplikację, „cake" kod frameworka, a „vendors" wspólne dodatki (klasy obce).

W ujęciu CakePHP elementy MVC reprezentują:

- Model - odzwierciedla konkretną tabelę bazy danych i jej relację z innymi tabelami. Zawiera reguły dotyczące typu przechowywanych danych stosowane podczas ich wstawiania i aktualizacji (wzorzec *ActiveRecord*). Model rozszerza klasę `AppModel`. Dla każdej tabeli bazy danych używanej w aplikacji musi istnieć jej model. Modele znajdują się w w katalogu „app/models". W pliku konfiguracyjnym znajdują się informacje o bazie. Dostępne bazy to: MySQL, PostgreSQL, SQLite, drivery do PEAR oraz ADO.
- Kontroler - obsługuje żądania serwera, pobiera dane wejściowe od użytkownika (w formie adresu URL i danych POST), stosuje reguły logiki biznesowej. Następnie używa modelu, aby czytać i zapisywać informacje z bazy danych lub innych źródeł. Ostatecznie kontroler wysyła dane wyjściowe do widoku w celu ich wyświetlenia. Kontroler rozszerza klasę `AppController`. Kontrolery znajdują się w katalogu „app/controllers", według konwencji nazwa_controller.php.
- Widok - szablon strony HTML ze zintegrowanym kodem PHP odpowiedzialnym za wyświetlenie danych wyjściowych aplikacji. Widoki przechowywane w podkatalogach „app/views" nazwanych odpowiednio do nazw obsługujących kontrolerów. Pliki widoków są nazwane tak, jak akcje kontrolera z rozszerzeniem .thtml. Istnieje możliwość stworzenia layoutu, który będzie wielokrotnie wykorzystany.

4.4.2. Symfony

Symfony jest zorientowanym obiektowo frameworkiem PHP5 opartym na modelu MVC, Rys. 8. Symfony jest inspirowana Ruby on Rails - patrz Rozdział 4.2.1 oraz Mojavi. Framework ma budowę modułową: projekt składa się z aplikacji, które dzielą się na moduły. Moduły zawierają operacje. Struktura katalogów jest narzucona z góry. Symfony używa języka YAML, przypominającego XML, do tworzenia plików konfiguracyjnych



Rysunek 8: Symfony

Komponenty modelu dzielą się na:

- warstwa abstrakcyjna bazy danych - funkcje obsługi specyficzne dla konkretnej bazy danych,
- warstwa dostępu do bazy - niezależna od bazy danych.

Konieczne jest stworzenie pliku .yml zawierającego podstawowe informacje o bazie danych. W kolejnym pliku .yml zapisana jest struktura bazy, dzięki czemu klasy modelu zostaną automatycznie stworzone. Symfony korzysta z bibliotek Propel do implementacji ORM oraz Creole jako abstrakcyjnej warstwy bazy.

Komponenty kontrolera dzielą się na:

- `Front Controller`, stanowi wejście do aplikacji, ładuje konfigurację i wybiera, która z akcji ma być wykonana. `Front Controller` jest generowany automatycznie,

- klasy `Action` dziedziczące z `sfActions` zgrupowane w modułach, zawierają metody implementujące konkretne akcje. Zawierają logikę aplikacji, korzystają z modeli oraz definiują dane dla widoków,

- obiekty typu `response, request, session` np. zwracające parametry żądania,

- filtry, czyli fragmenty kodu wykonywane przed lub po akcji, obsługujące np. walidację.

Widok składa się z dwóch odrębnych części:

- warstwa prezentacji: oparta jest o wzorzec *Dekorator* - szablon w HTML z dodatkiem PHP zwracany przez akcję jest dekorowany przez domyślny szablon główny znajdujący się w pliku layout.php (zawierający identyczny dla wielu stron layout),

- konfiguracja widoku w pliku view.yml.

Porównanie Poniżej przedstawione zostaną cechy obydwu frameworków.
CakePHP

- aktualna ale nie do końca kompletna dokumentacja,

- łatwa instalacja,

- prosta struktura,

- wsparcie dla Ajax,

- walidacja danych.

Symfony

- niekompletna dokumentacja,

- uciążliwa instalacja,

- skomplikowana struktura katalogów,

- rozbudowany framework,

- tylko PHP5,

- własny ORM (Propel i Creole).

4.5. .NET

4.5.1. ASP.NET

ASP.NET nie promuje wzorca *MVC* w takim zakresie jak J2EE, opisany w Rozdziale 4.6.2, choć implementacja *MVC* jest możliwa, a jej sposób opisany w dokumentacji ASP.NET, patrz [18].

Widok to strona .aspx lub .ascx zawierająca kod HTML, która jest odpowiedzialna za wyświetlenie danych.

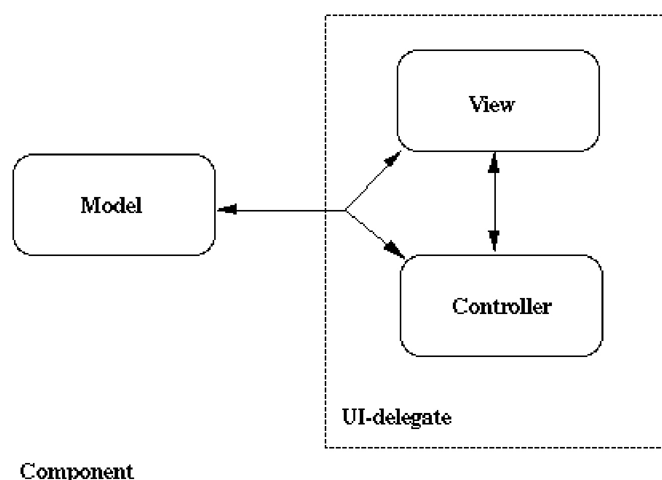
Kontroler wykonuje odpowiednie zadania w odpowiedzi na żądania widoku. Obowiązki kontrolera zostały podzielone. Generacja oraz przekazywanie żądań jest zadaniem frameworka (klas `Page` oraz `Control`). Obsługa żądań została przeniesiona do klas będących tzw. *Code Behind*, związanych ze stroną, znajdujących się w plikach .aspx.cs lub .aspx.vb. Klasa ta dziedziczy po `System.Web.UI.Page`. Implementacja kontrolera w ASP.NET przypomina wzorzec projektowy *Page Controller*, w którym dla każdej strony istnieje moduł spełniający funkcje jej kontrolera (ta sama strona lub oddzielny obiekt odpowiadający danej stronie). Możliwa jest jednak również implementacja wzorca *Front Controller*, charakterystycznego dla implementacji *MVC* dla J2EE.

Klasa modelu nie jest wymagana. Funkcjonalność modelu może być zaimplementowana w osobnej klasie lub należeć do kontrolera. Obsługę modeli wspiera biblioteka ADO.NET. Nie jest wymagane utrzymywanie otwartego połączenia z bazą danych. Aplikacje utrzymują połączenie tylko na czas odczytywania lub zapisywania danych. Dane pobrane z bazy są przechowywane w obiekcie `DataSet` pełniącym role bufora. ADO.NET obsługuje większość baz danych

4.6. Java

4.6.1. Swing

Swing to biblioteka komponentów GUI przeznaczona do budowy graficznych interfejsów użytkownika dla aplikacji komercyjnych. Swing korzysta z architektury *quasi-MVC* (ang. *seperable model architecture*), przedstawionej na Rys. 9.



Rysunek 9: Model *MVC* w Swingu

Widok oraz kontroler każdego komponentu zostały połączone w jeden obiekt: *UI object* lub *delegate UI object*. W ten sposób model danych jest odseparowany, co umożliwia programowanie w kategoriach modelu. Zadania uwidaczniania i interakcji są delegowane przez komponent do odpowiedniego obiektu UI. Modele w Swingu są realizowane jako interfejsy. Można wyróżnić dwa typy modeli:

- modele GUI - interfejsy definiujące wizualne stany komponentów, np. zaznaczenie elementu na liście,
- modele danych - interfejsy reprezentujące dane takie jak np. elementy listy.

Dostęp do modeli jest zapewniony poprzez metody `getModel()` oraz `setModel()` w klasach komponentów. Istnieje możliwość tworzenia własnego modelu. Wszelkie zmiany są dokonywane w modelu danych, a o każdej zmianie model musi powiadamiać zainteresowanych słuchaczy. Budowa klas w Swingu gwarantuje, że zdarzenia te będą obsłużone przez odpowiednie obiekty UI.

Dobrym przykładem *MVC* w Swingu jest konfigurowalny wygląd *pluggable look&feel* - prezentacja(wygląd) i obsługa zdarzeń realizowana przez komponent może ulec zmianie w czasie działania programu.

4.6.2. J2EE

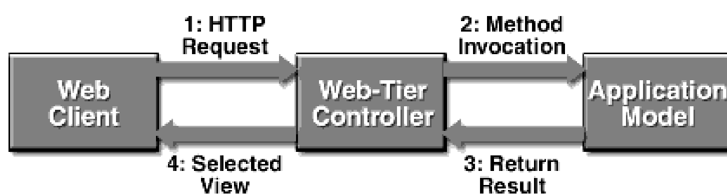
MVC jest rekomendowanym wzorcem projektowym dla Java 2 Platform Enterprise Edition - J2EE. Realizacja architektury *MVC* na platformie J2EE angażuje następujące technologie:

- komponenty typu kontroler są realizowane jako serwlety Java,
- komponenty widoku to strony Java Server Pages JSP(Model 2) ,
- komponenty modelu są realizowane poprzez:
  - interfejs JDBC,
  - technologię ORM - odwzorowanie obiektów z Javy na struktury relacyjne. Można zastosować Hibernate, Oracle TopLink, Enterprise Java Beans EJB, Java Data Objects JDO.

Dokumentacja J2EE wymienia dwa typy architektury: *Model 1* oraz *Model 2*. W przypadku architektury typu *Model 1* strona JSP jest odpowiedzialna zarówno za odbieranie żądań, jak i prezentowanie wyników. *Model 2* jest rozbudową *Modelu 1* o serwlet, który pełni funkcję kontrolera. Serwlet ten przetwarza żądania klienta (przeglądarki), buduje obiekty JavaBeans, które przekazuje odpowiednim stronom JSP. Strona JSP odpowiada jedynie za prezentację i nie zawiera sama w sobie kodu logiki programu. JSP jedynie odbiera obiekty JavaBeans wyprodukowane przez serwlet kontrolera.

Kontroler odwzorowuje interakcje z widokiem (w przypadku aplikacji webowych są to żądania HTTP: GET i POST) na operacje modelu i wybiera odpowiedni widok. Stosowany jest tu wzorzec *Front Controller*, Rys. 10, który centralizuje przetwarzanie żądań i wybór widoków w jeden komponent.

Schemat przykładowej aplikacji przedstawiono na Rys. 11:



Rysunek 10: Front Controller

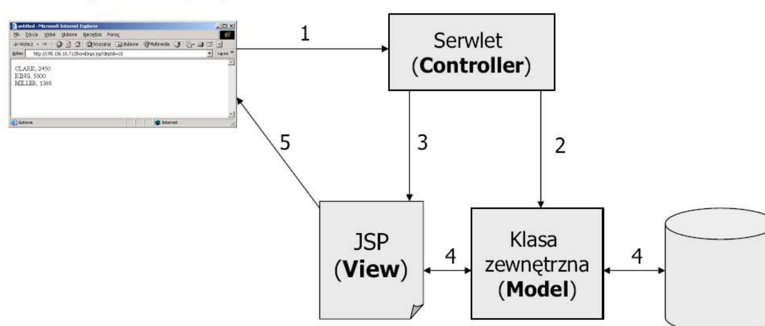
1 Przeglądarka wysyła żądanie uruchomienia serwletu - kontrolera.

2 Serwlet analizuje żądanie i tworzy obiekty klas zewnętrznych JavaBeans realizującyh funkcje modelu. W obiektach tych wywoływane są funkcje logiki biznesowej.

3 Serwlet przekazuje sterowanie do odpowiedniej strony JSP, realizującej funkcje widoku.

4 JSP pobiera dane z obiektów modelu przygotowanych przez kontroler. Obiekty te mogą udostępniać dane pobrane z bazy danych.

5 JSP generuje wynikowy dokument dla użytkownika.



Rysunek 11: Schemat przykładowej aplikacji na platformie J2EE

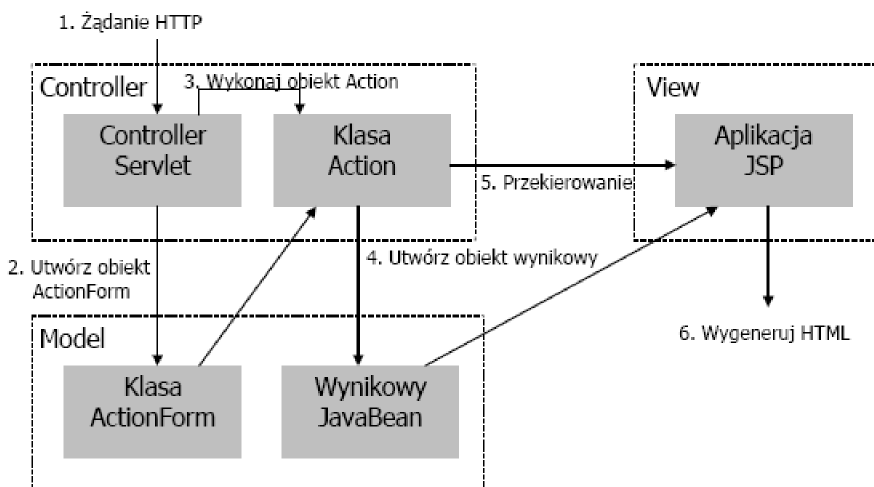
Struts Struts jest popularną implementacjąą J2EE powstałą w ramach projektu Apache/Jakarta, ułatwiającą tworzenie aplikacji webowych zgodnie z paradygmatem *Model 2 MVC*. Architektura Struts została pokazana na Rys. 12.

Warstwa prezentacji budowana jest za pomocą stron JSP i standardowych znaczników umożliwiających interakcję z komponentami JavaBean przenoszącymi informacje. Widok jest odpowiedzialny za obsługę internacjonalizacji oraz automatyczną walidację formularzy. Możliwe jest wykorzystanie specjalizowanych bibliotek znaczników JSP, np. Struts Bean. Można także budować widoki w oparciu o XLST oraz Velocity.

Model reprezentuje stan aplikacji jako zbiór instancji JavaBeans. Można korzystać z JDBC, EJB, Object Relational Bridge.

Serwlet-kontroler odpowiedzialny jest za:

- przetwarzanie żądań użytkowników,



Rysunek 12: Architektura aplikacji Apache Struts

- wybór właściwego widoku i przekazanie go użytkownikowi za pomocą odwzorowania `ActionMapping`,
- parsowanie pliku konfiguracyjnego i inicjalizacja aplikacji.

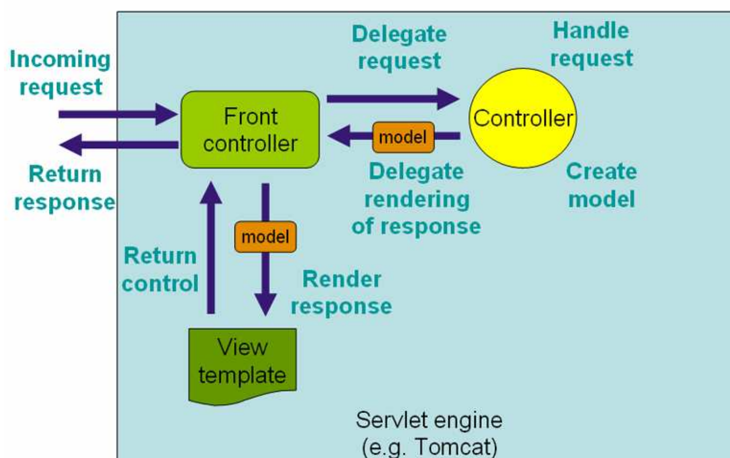
Podstawowym elementem Struts jest `ActionServlet`, który zarządza aplikacją na podstawie konfiguracji zapisanej w pliku struts-config.xml (zapisane są tam ścieżki nawigacyjne oraz referencje do wszystkich obiektów klas `Action` i `ActionForm`, reprezentującej dane z formularzy). Gdy serwlet odbiera żądanie użytkownika, tworzy obiekt klasy obsługi żądań, dziedziczący z klasy `Action`. Klasa `Action` odczytuje dane wejściowe, łączy się z bazą danych i uruchamia logikę biznesową w celu odczytania danych wyjściowych. Dane wyjściowe są umieszczane jako zbiór komponentów JavaBean. Następnie sterowanie zostaje przekazane do wybranej strony JSP.

Spring Spring jest szkieletem wytwarzania aplikacji w języku Java dla platformy J2EE. Spring udostępnia *Spring's Web MVC* do wytwarzania aplikacji webowych w oparciu o wzorzec *MVC*. Na Rys. 13 przedstawiono przepływ żądania HTTP.

Podstawowym składnikiem Spring MVC jest `ServletDispatcher`, na Rys. 13 przedstawiony jako `Front Controller`, który odpowiada za przyjmowanie żądań z zewnątrz i przekierowanie ich do właściwych kontrolerów. Każdy `ServletDispatcher` posiada swój kontekst aplikacji webowej, umożliwiający dostęp do różnych beanów: .

- *Controllers* - kontrolery ze wzorca *MVC*,
- *View resolvers* - komponenty umożliwiające mapowanie nazw na widoki,
- *Locale resolvers* - komponenty odpowiedzialne za internacjonalizację widoków.

Konfiguracja znajduje się w pliku .xml.



Rysunek 13: Przepływ żądań w Spring Web MVC

Kontrolery w Springu to interfejsy, które interpretują dane wejściowe i zamieniają w odpowiedni model reprezentowany użytkownikowi w postaci widoku. Interfejs kontrolera zwraca obiekt typu `ModelAndView`. Składa się on z nazwy widoku oraz modelu typu `Map`, zawierającego nazwy beanów oraz powiązane z nimi obiekty. Specjalnym typem kontrolera w Springu są tzw `Command Controllers`, umożliwiające mapowanie parametrów z żądań HTTP w obiekty modelu. Pełnią one podobną rolę jak `ActionForm` w Struts, ale nie trzeba implementować specyficznych dla frameworka interfejsów.

Spring MVC nie jest związany ściśle z JSP w warstwie widoku, wspiera wiele innych technologii: szablony Velocity, XSLT, Tiles, Excel, PDF. Spring umożliwia integrację ze Struts, WebWork, Tapestry. Trwają prace nad integracją Springa z JSF.

Java Server Faces Java Server Faces to zrąb aplikacji Java do tworzenia interface'u użytkownika dla aplikacji webowych.

Głównym zadaniem JSF jest dostarczenie modelu komponentów do tworzenia interfejsu użytkownika widoku. Komponenty są konfigurowalne, rozszerzalne, niezależne od technologii prezentacji. JSF nie wymaga JSP jako technologii widoku, ale JSP jest domyślną technologią do renderowania komponentów JSF. JSF dostarcza implementację kontrolera w postaci konfigurowalnego serwletu `FacesServlet`. JSF nie wspiera tworzenia modelu. Udostępnia jedynie mechanizmy wiążące obiekty modelu z pozostałymi komponentami aplikacji.

Porównanie Poniżej przedstawione zostaną cechy opisanych frameworków J2EE.
Wszystkie z frameworków zapewniają walidację oraz internacjonalizację.

Zalety	Wady
• Popularność, • Dużo informacji, • Dobra biblioteka tagów HTML, • Duża ilość IDE.	• Obsługa *ActionForm*, • Testy - tylko StrutsTest-Case.

Tablica 1: Struts - zalety i wady

Zalety	Wady
• Popularność, • Integracja z JSP, Tiles, Velocity, FreeMarker, Excel, PDF, XSL, • Łatwość testowania, • Integracja z JSF, Struts.	• Skomplikowana konfiguracja XML, • Zbyt elastyczny, • Średnio popularny, • Mała wybór IDE.

Tablica 2: Spring MVC - zalety i wady

Zalety	Wady
• Standard J2EE, • Szybki, łatwy w pracy, • Prosty kontroler oraz konfiguracja aplikacji.	• Bałagan w tagach, • Brak walidacji po stronie klienta, • Różne źródła implementacji, • Młoda techonologia.

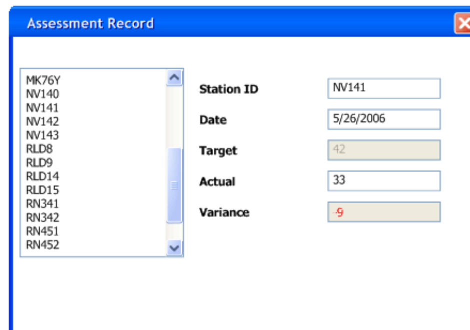
Tablica 3: JSF - zalety i wady

5. MVC w przykładach

5.1. Przykłady - ogólna koncepcja

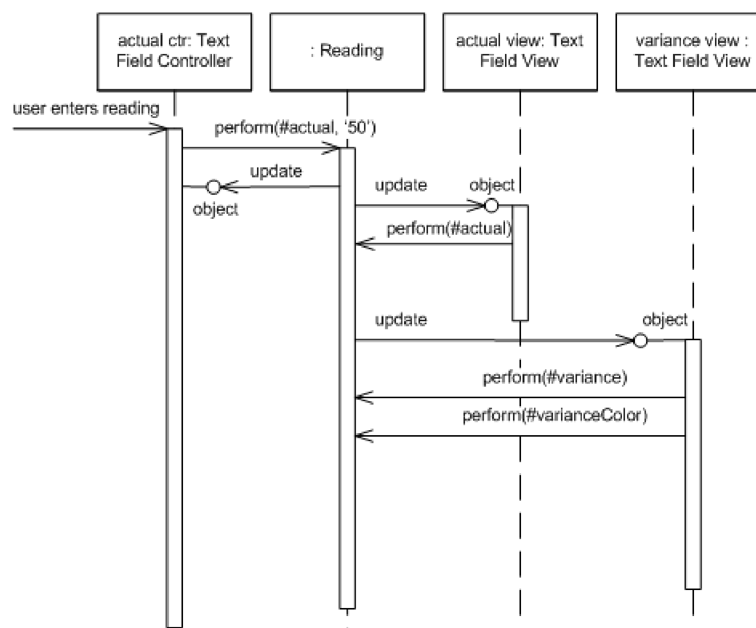
5.1.1. Aplikacja GUI

Przykład implementacji wzorca *MVC* w aplikacji typu GUI zostanie podany za Martinem Fowlerem, patrz [23].



Rysunek 14: Przykładowa aplikacja

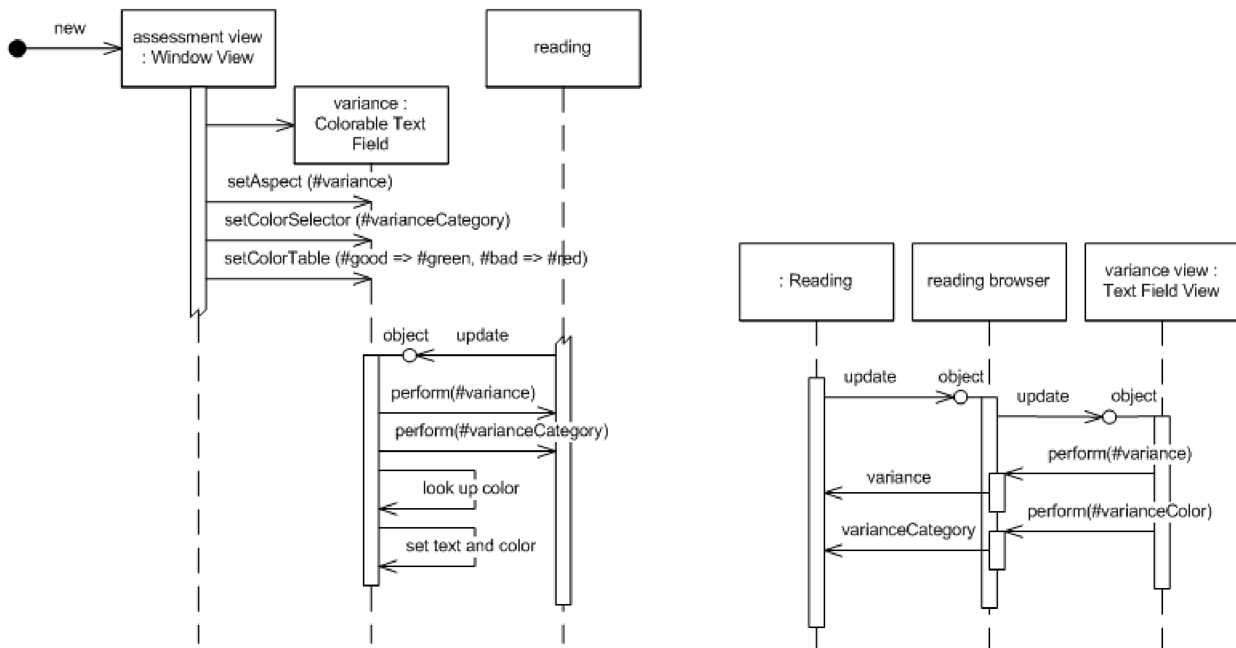
Funkcjonalność przykładowej aplikacji, patrz Rys. 14, jest następująca: użytkownik może wybrać stację, podać datę oraz wartość aktualną. System wylicza wariancję od wartości docelowej, a w przypadku zbyt niskiej wartości, pole wariancji jest wyróżnianie innym kolorem.



Rysunek 15: Diagram sekwencji

Gdy użytkownik wpisuje nową wartość aktualną, patrz Rys. 15, kontroler pola tekstowego zmienia pole obiektu `Reading` reprezentującego model. Następnie model powiadamia zarejestrowane pola tekstowe widoku o zmianie. Pole tekstowe widoku wywołuje metodę na obiekcie

modelu i ustawia swoją wartość zgodnie ze zwróconą przez metodę wartością. Kontroler nie modyfikuje widoku. Za aktualizację pól tekstowych widoku odpowiada mechanizm wzorca *Obserwator*, opisany w Rozdziale 3.1. Jest to przykład zastosowania Modelu Aktywnego z Rozdziału 2.2.1.



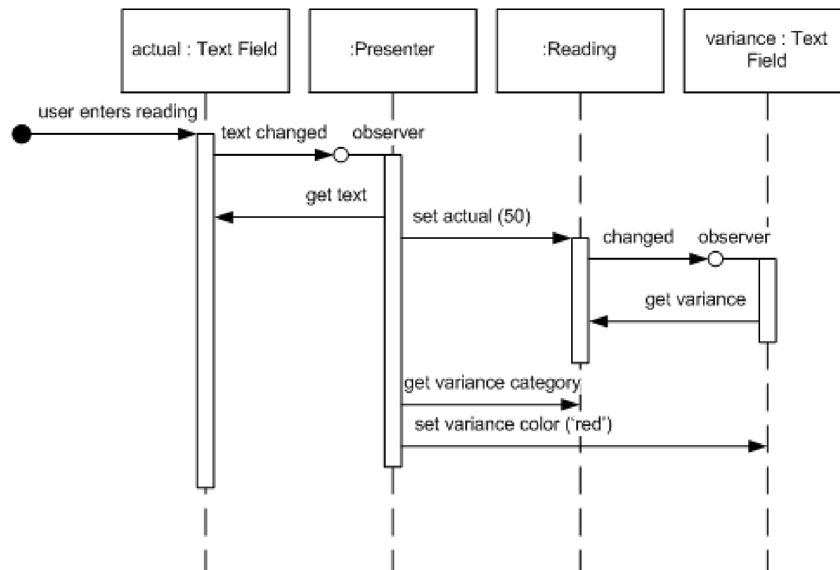
Rysunek 16: Presentation Model - diagram sekwencji

Nie uwzględniona dotychczas konieczność zmiany koloru pola wariancji w zależności od wartości docelowej wymaga kooperacji logiki modelu dziedziny *Domain Model* oraz modelu prezentacji *Presentation Model*. Kwestia zakwalifikowania modelu prezentacji do wybranego komponentu *MVC* staje się problematyczna, ponieważ nie należy ani do warstwy modelu jak i widoku w klasycznym *MVC*. Możliwe rozwiązania zostały przestawione na Rys. 16. Jednym z rozwiązań może być utworzenie klasy dziedziczącej po polu tekstowym widoku i dodanie jej nowej funkcjonalności. Innym rozwiązaniem jest utworzenie dodatkowego modelu odpowiedzialnego za logikę warstwy prezentacji, który jest wciąż niezależny od widoku.

Rozwinięciem wzorca *MVC* jest *MVP Model View Presenter*, patrz Rys. 17. W tym wypadku to pola tekstowe reprezentujące widok (widoki nie posiadają swoich kontrolerów) odbierają żądania użytkownika i przekazują je do obiektu `Presenter`. `Presenter` podejmuje odpowiednie działania w odpowiedzi na akcje użytkownika i aktualizuje model. Widoki są uaktualniane poprzez mechanizm wzorca *Obserwator*, choć istnieje możliwość bezpośredniej manipulacji widokiem przez `Presenter`. Akcje reprezentujące logikę widoku mogą być rozdzielona pomiędzy widok a `Presenter` (Fowler wprowadza nowy wzorzec, tzw. *Supervising Controller*) bądź być obsługiwane tylko przez `Presenter` (cytując za Fowlerem - wzorzec *Passive View*).

5.1.2. Aplikacja webowa

Przykład realizacji wzorca MVC w aplikacji webowej pochodzi z artykułu Davida J. Andersona, patrz [24]. Autor wprowadza następującą interpretację wzorca *MVC*: zadaniem kontrolera jest



Rysunek 17: MVP - diagram sekwencji

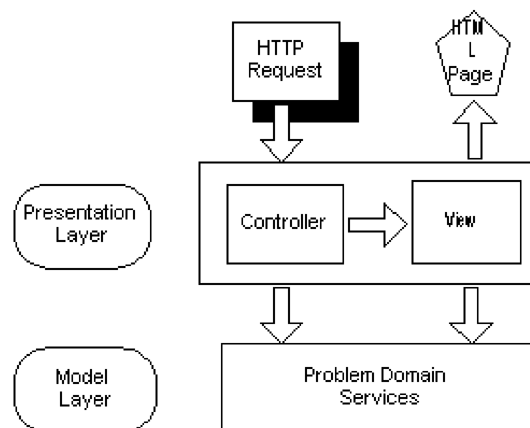
przetwarzanie żądań użytkownika, aktualizowanie modelu i wybór odpowiedniej strony - widoku w odpowiedzi na akcje użytkownika. Kontroler oraz widok należą do warstwy prezentacji, tzw. *Presentation Layer* w architekturze trójwarstwowej, patrz Rys. 18(a). Model nie powiadamia widoków o zmianach, natomiast widoki podczas inicjalizacji same pobierają dane z modelu. Na Rys. 18(b) znajduje się diagram stanów aplikacji webowej.

Przykładowa aplikacja webowa ma na celu zalogowanie użytkownika. Użytkownik wprowadza login i hasło. W zależności od wyniku weryfikacji danych jest przekierowany na główną stronę, bądź powraca do strony logowania. Użytkownik może maksymalnie trzykrotnie podać błędne dane. Diagram nawigacji pomiędzy stronami znajduje się na Rys. 19(a), natomiast diagram stanów na Rys. 19(b). Budując aplikację w oparciu o architekturę *MVC*, Rys. 20, można wyróżnić trzy widoki:

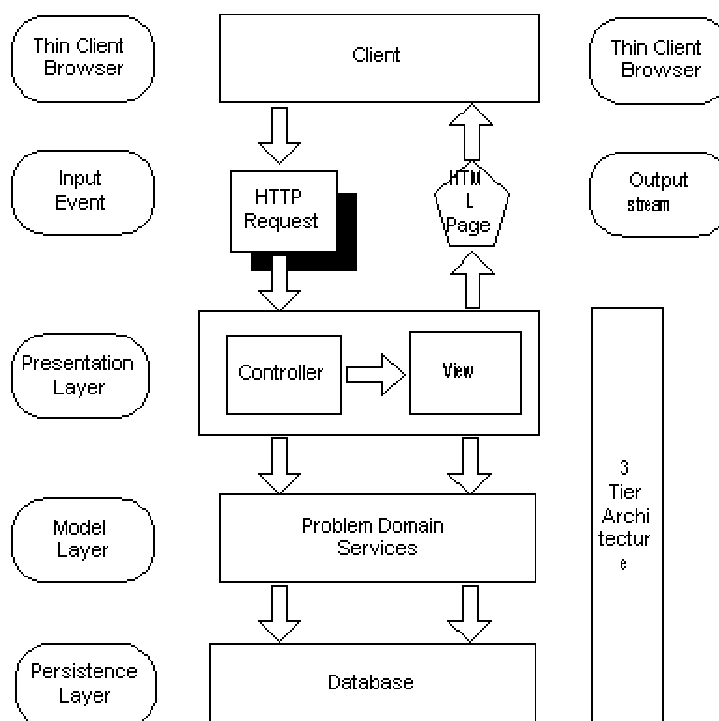
- `LoginRequestView`,
- `AccessDeniedView`,
- `MainApplicationView`.

Kontroler `LoginController` wywołuję odpowiednie funkcje logiki biznesowej w odpowiedzi na akcje użytkownika.

Kolejny krok polega na stworzeniu diagramu klas, patrz Rys. 21. Kontrolery oraz stany odpowiadające stronom HTML są reprezentowane przez klasy. Klasa `HTTPRequest` odpowiada za przyjmowanie żądań od użytkownika i przekazywanie ich do kontrolera. Klasa `LoginSession` reprezentuje model i jest wykorzystana przez kontroler do weryfikacji danych użytkownika.

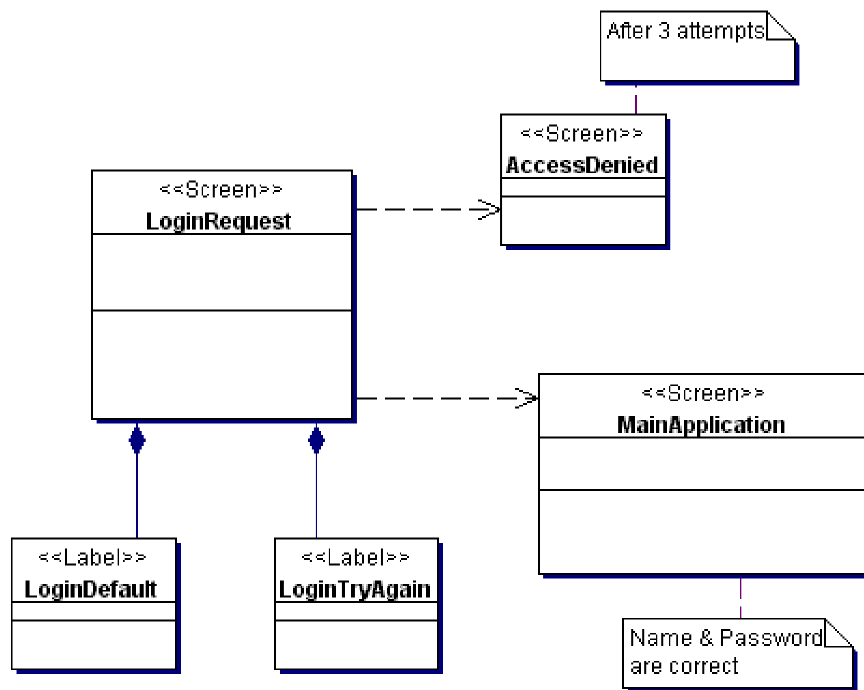


(a) Architektura trójwarstwowa

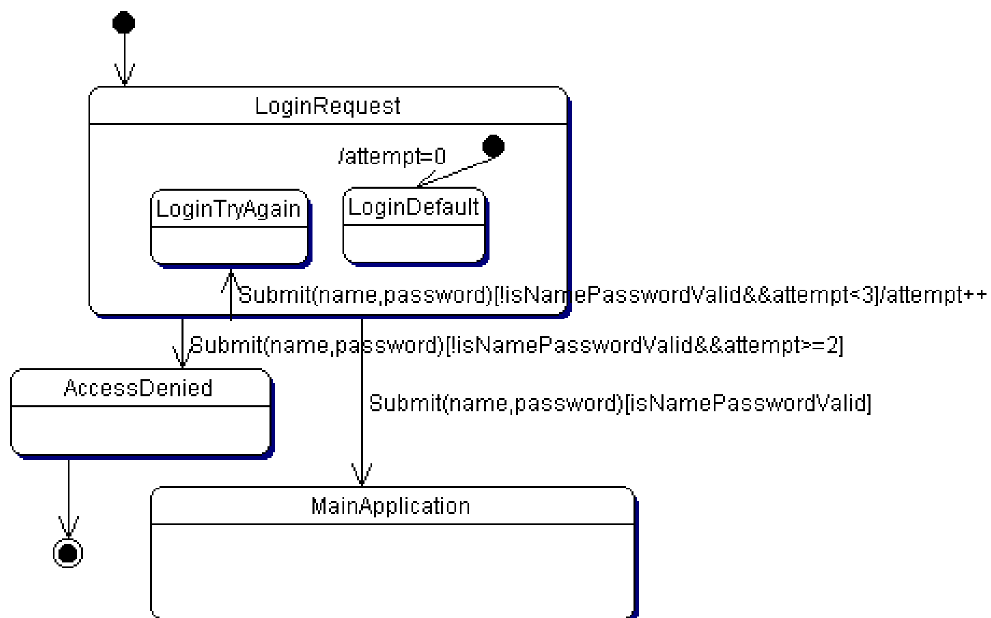


(b) Diagram stanów

Rysunek 18: MVC w aplikacji webowej

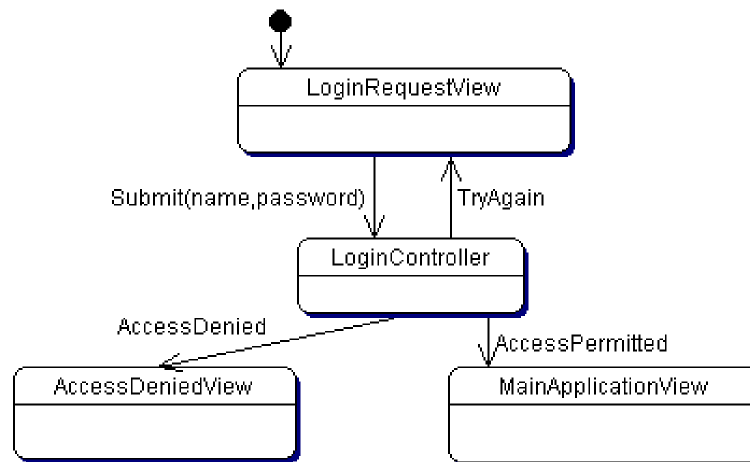


(a) Nawigacja pomiędzy stronami

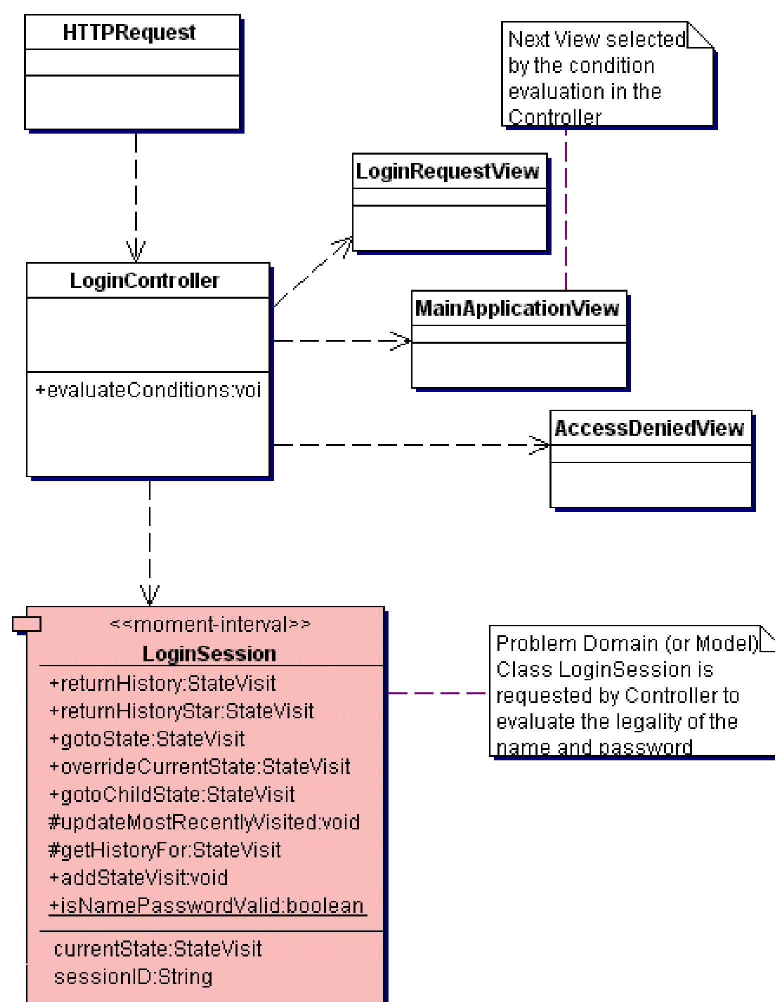


(b) Diagram stanów w przykładowej aplikacji

Rysunek 19: Przykładowa aplikacja



Rysunek 20: Architektura MVC w przykładowej aplikacji

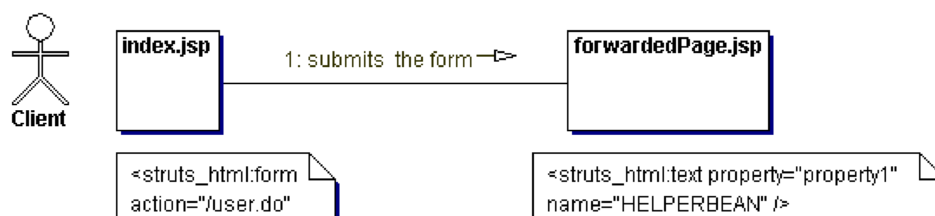


Rysunek 21: Diagram klas

5.2. Przykłady - konkretne frameworki

5.2.1. Struts

Realizacja wzorca *MVC* przy użyciu Struts zostanie przedstawiona na przykładzie aplikacji o bardzo prostej funkcjonalności, patrz Rys. 22: klient zatwierdza formularz, żeby przejść do kolejnej strony. Autorem przykładu jest Jean-Michel Garnier, patrz [25].



Rysunek 22: Funkcjonalność

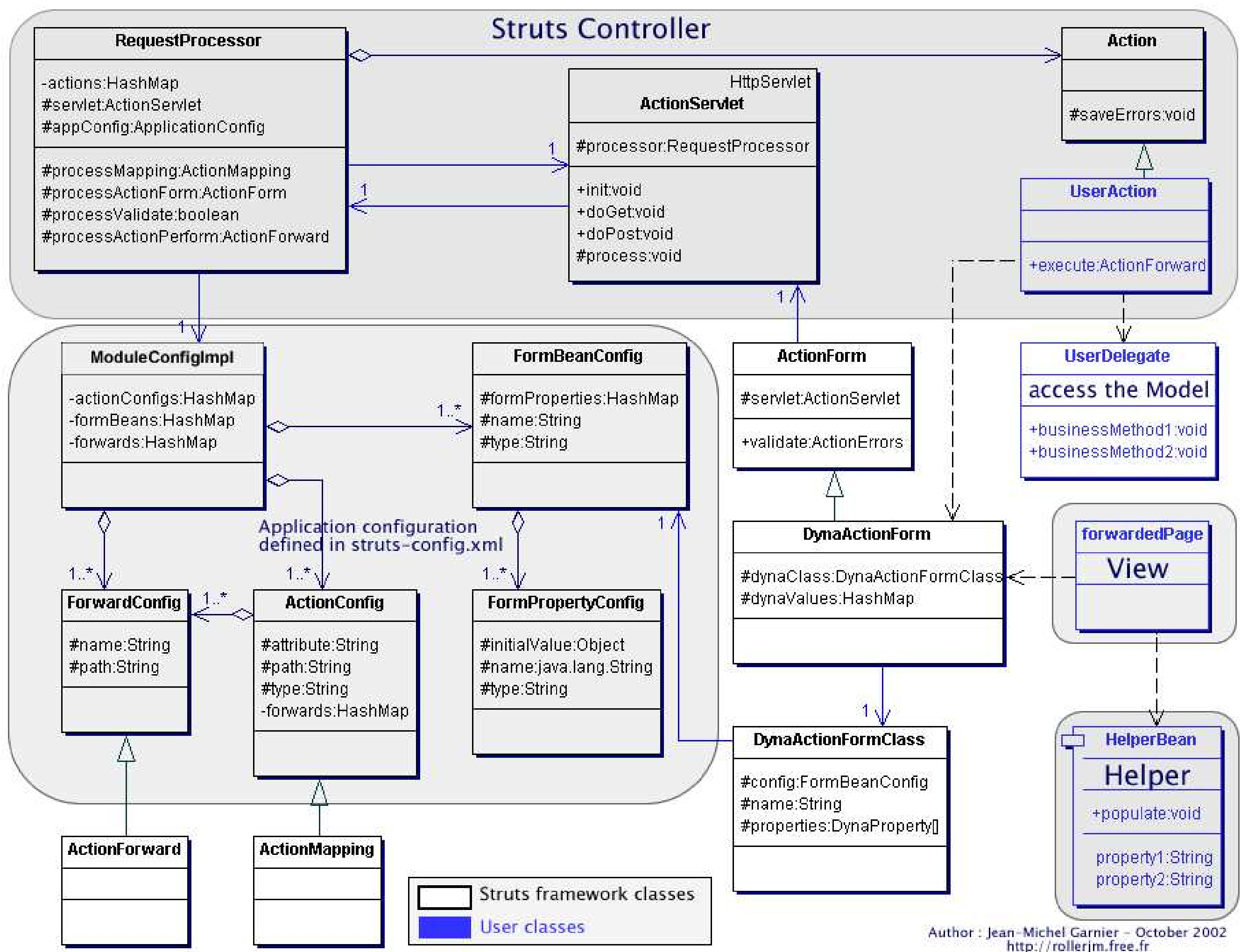
Diagram klas znajduje się na Rys. 23.
Konfiguracja - `ModuleConfigImpl` zawiera informacje dotyczące konfiguracji aplikacji. Podczas inicjalizacji plik konfiguracyjny jest parsowany w celu wysłania informacji do następujących beanów:

- Zarządzanie mapowaniem:
  - `ActionConfig` zawiera informacje dotyczące mapowania żądań na klasy Action. Przykład:
    `<action path="/user" type="org.xxx.UserAction" name="userForm" scope="session">`
  - `ForwardConfig` zawiera informacje dotyczące nawigacji między stronami. Przykład:
    `<forward name="next" path="/forwardedPage.jsp" />`.
- Zarządzanie beanami `ActionForm`:
  - `FormBeanConfig` zawiera definicję beana `ActionForm`. Przykład:
    `<form-bean name="userForm" type="org.apache.struts.action.DynaActionForm" />`.
  - `FormPropertConfig` to bean reprezentujący informacje elementu `<form-property>`.

Kontroler - klasa `ActionServlet` jest sercem aplikacji. Otrzymuje żądania od przeglądarki i przesyła w zależności od informacji zawartych w pliku konfiguracyjnym. Klasy dziedziczące po `Action` zawierają logikę biznesową. Instancje klasy `Action` sterują przekazywaniem danych z formularzy do modelu.

Zarządzanie formularzami HTML - klasy `ActionForm` reprezentują stan aplikacji. Istnieje możliwość zastosowania komponentów dynamicznych `DynaActionForm`, których definicja jest w całości umieszczona w pliku konfiguracyjnym.

Klasy użytkownika - klasa `HelperBean` to *Value Object Bean*, który przechowuje dane z formularzy (ma na celu zmniejszenie ruchu w sieci, ponieważ informacja zawarta w atrybutach jest przeniesiona na stronę klienta). Klasa `UserAction` dziedziczy po klasie `Action` i implementuje metodę `execute()` zawierającą logikę aplikacji. Klasa `UserDelegate` oddziela komponenty



Rysunek 23: Przykładowa aplikacja

warstwy biznesowej od kodu oraz enkapsuluje dostęp do modelu. `ForwardedPage.jsp` to docelowa strona .jsp, która reprezentuje widok.

Diagram sekwencji znajduje się na Rys. 24. Opis komunikatów:

1 Klient zatwierdza formularz, zostaje wywołana metoda `doPost`.

1.1 Kontroler Struts `ActionServlet` oddelegowuje żądanie do `RequestProcessor`.

1.1.1 `RequestProcessor` zwraca beana `ActionForm`.

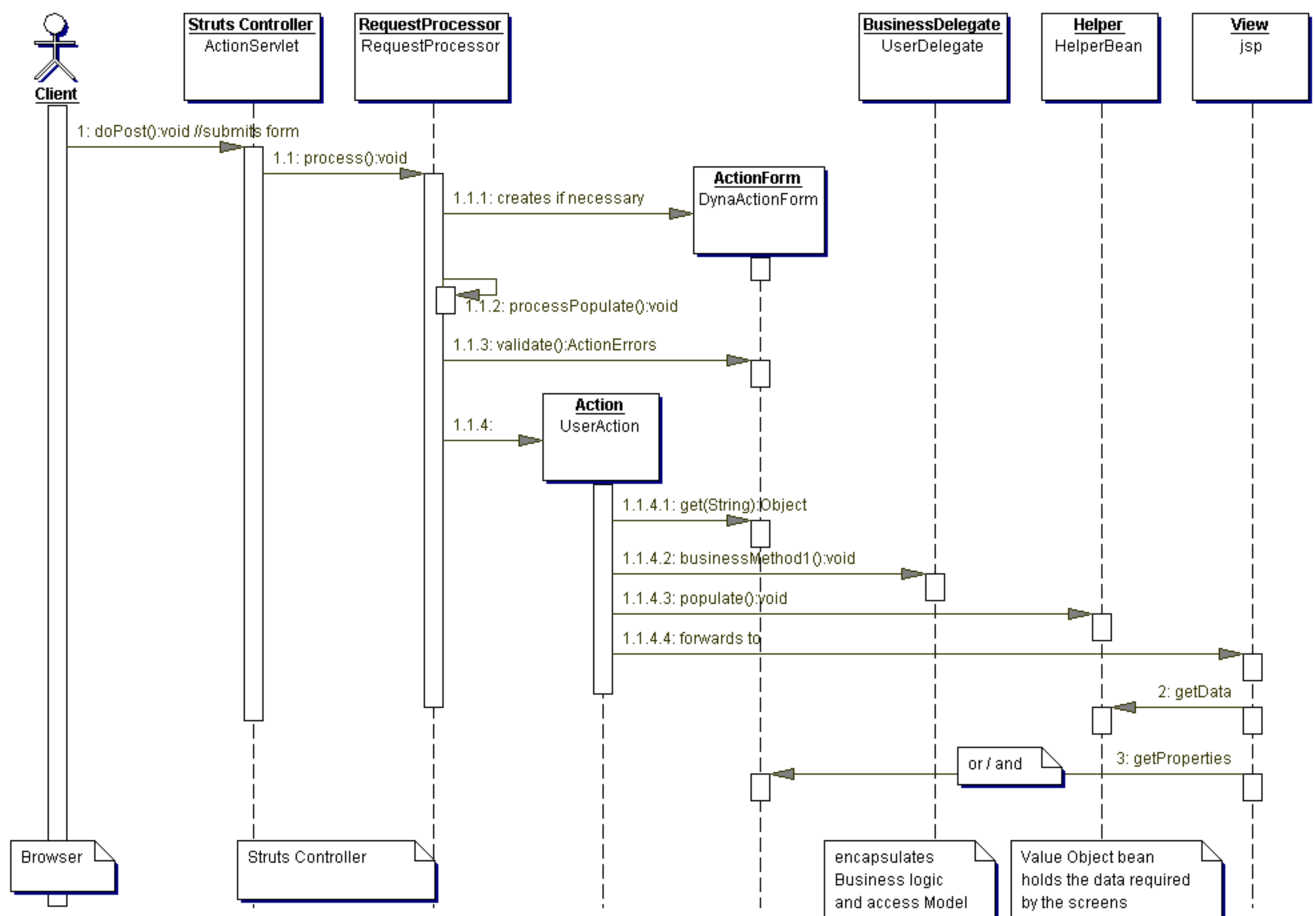
1.1.2 `RequestProcessor` wypełnia beana danymi z formularza.

1.1.3 `RequestProcessor` przeprowadza walidację danych i generuje komunikaty o błędach.

1.1.4 Tworzy instancję `UserAction` do przetworzenia żądania za pomocą przeciążonej metody `execute()`.

1.1.4.1 Pobranie danych z beana `UserActionForm` przy pomocy metody `getProperties`.

1.1.4.2 Wywołanie metod biznesowych.



Rysunek 24: Diagram sekwencji

1.1.4.3 Wypełnienie danymi *Value Object Beana*.

1.1.4.4 Przekierowanie do strony docelowej wyspecyfikowanej w pliku konfiguracyjnym.

2 Pobranie danych z `HelperBean`.

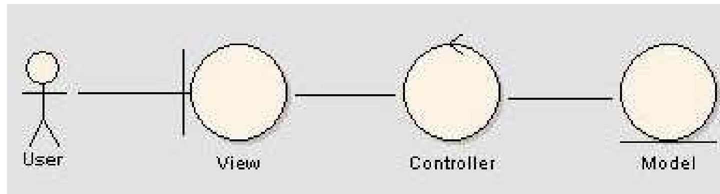
3 Pobranie danych z `ActionForm`.

5.2.2. ASP.NET

Realizacja wzorca MVC w ASP.NET została zaczerpnięta z przykładu pochodzącego z [26], którego autorem jest Shams Mukhthar. Autor wprowadza dodatkowe elementy diagramu UML, patrz Rys. 25, w celu wyróżnienia głównych obiektów występujących w systemie:

- `<<entity>>` reprezentują dane, odpowiednik Modelu. Mogą się porozumiewać tylko z obiektami `<<control>>`.
- `<<boundary>>` reprezentują wyjście systemu, odpowiednik Widoku. Mogą się porozumiewać z aktorami oraz obiektami `<<control>>`.

- `<<control>>` reprezentują logikę biznesową. Mogą się porozumiewać z obiektami `<<control>>`, `<<entity>>, <<boundary>>`.



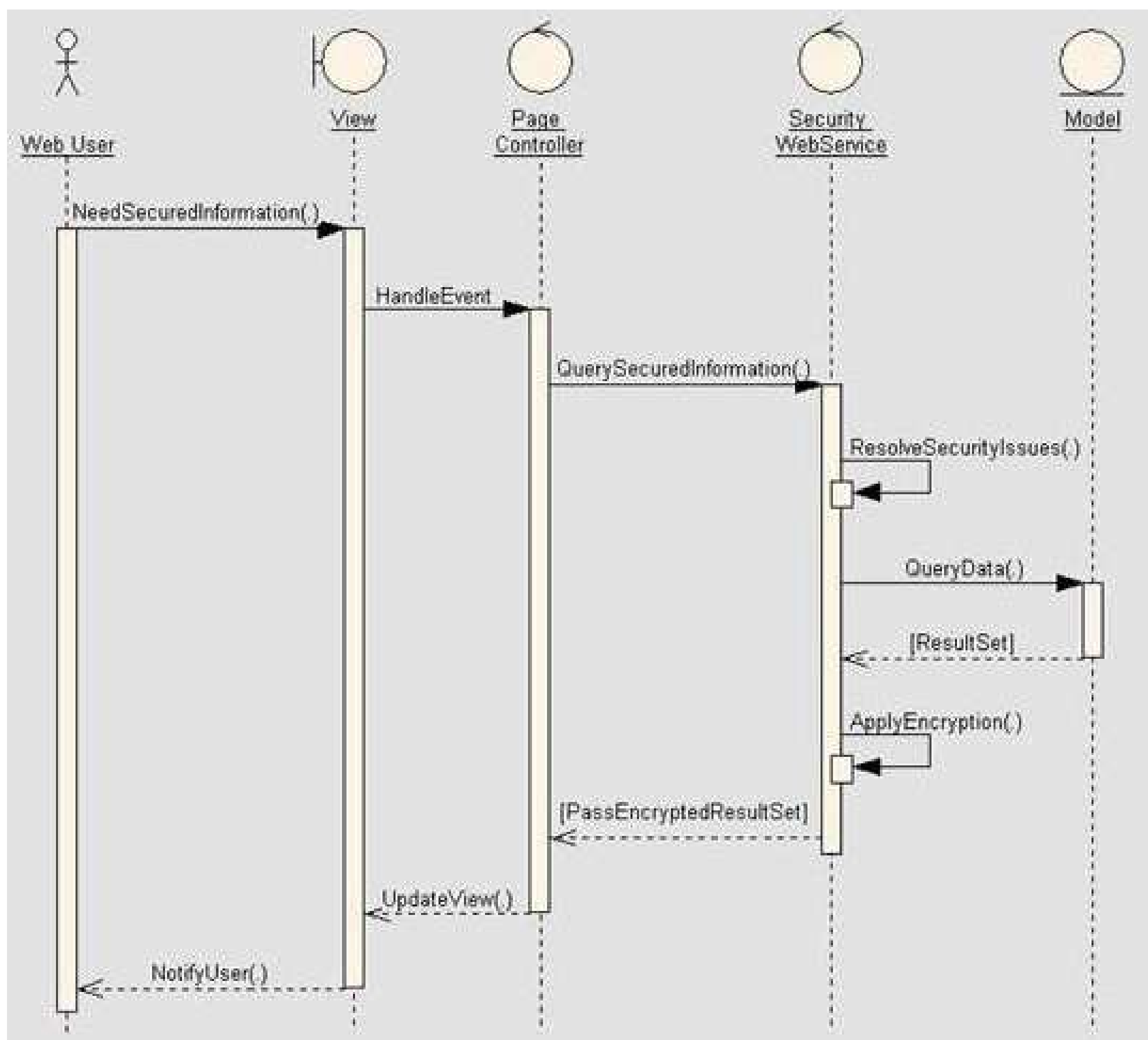
Rysunek 25: MVC - Robustness Analysis diagram

Wzorzec MVC w ASP.NET został opisany w Rozdziale 4.5.1. W ASP.NET, patrz Rys. 26, widok zależy w sposób pasywny od modelu. To kontroler realizujący wzorzec *PageController* odpowiada za akcje użytkownika wykonane na widoku i aktualizuje model. Można także wprowadzić dodatkowe kontrolery wykonujące bezpośrednio działania na modelu i zwracające wynik do *PageController'a*.

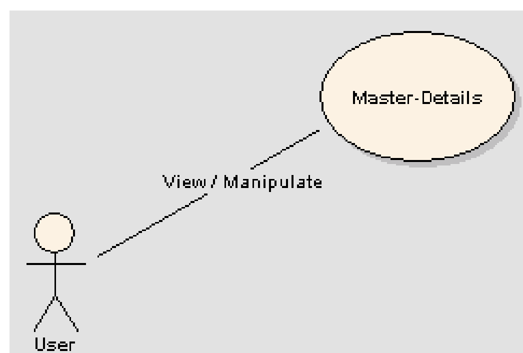
W przykładowej aplikacji, patrz Rys. 27, użytkownik będzie miał możliwość wyboru towaru w górnej tabeli, natomiast w tabeli dolnej zostaną wyświetlone jego szczegóły. Przykładowy layout znajduje się na Rys. 28. Na diagramie na Rys. 29 został przedstawiony przykładowy scenariusz. `MasterGridView` jest wykorzystany do wybrania towaru. O akcji użytkownika zostaje powiadomiony `MasterUserControl`, który następnie powiadamia słuchacza `MainController`, a ten z kolei powiadamia swojego słuchacza - obiekt `DetailsUserControl`. Wszystkie kontrolery mają dostęp do modelu reprezentowanego przez obiekt `DataAccessGateway`, który używa klasy `MsSqlDataLinkiAdapter` do nawiązania połączenia z bazą.

Kontrolery `MasterController` oraz `DetailsController` odpowiadają za uaktualnianie odpowiednich widoków, natomiast `MainControl` pośredniczy w wymianie informacji między nimi. Na Rys. 30 przedstawiono interakcje między tymi obiektami. Zarówno `MasterController` jak i `MainControl` implementują interfejs powiadamiania, natomiast `DetailsController` oraz `MainControl` implementują interfejs nasłuchu.

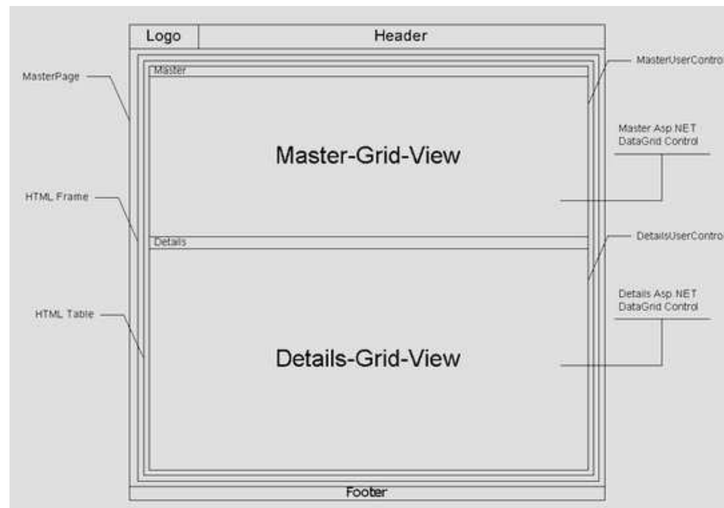
Na Rys. 31 znajduje się diagram klas analizowanego systemu.



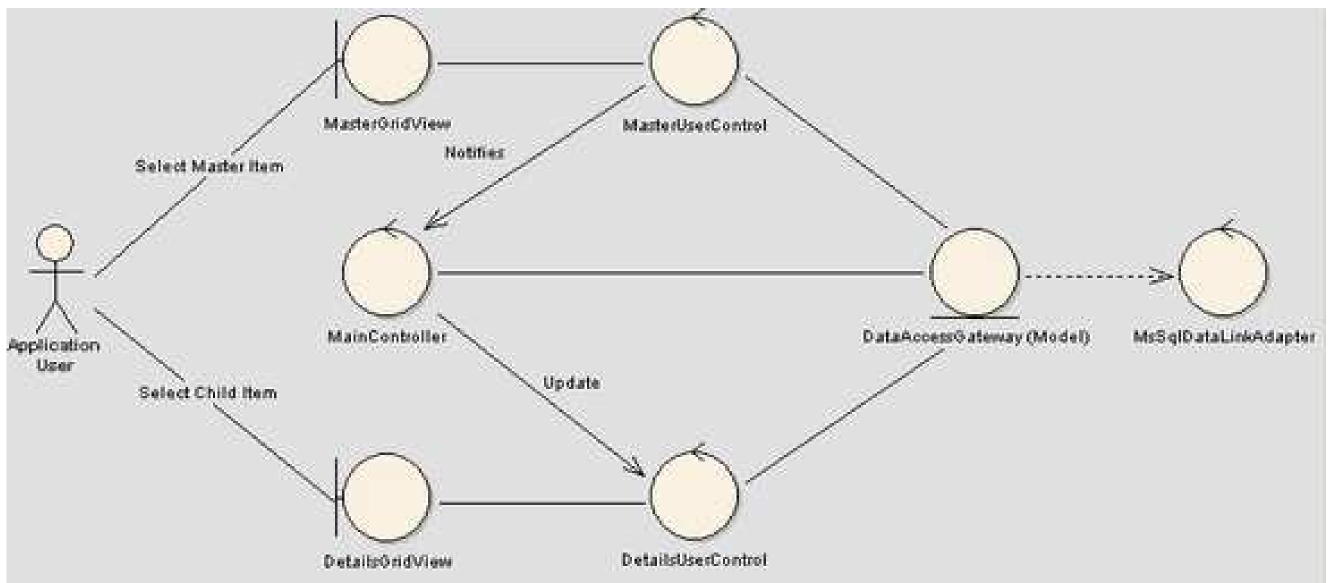
Rysunek 26: Diagram sekwencji



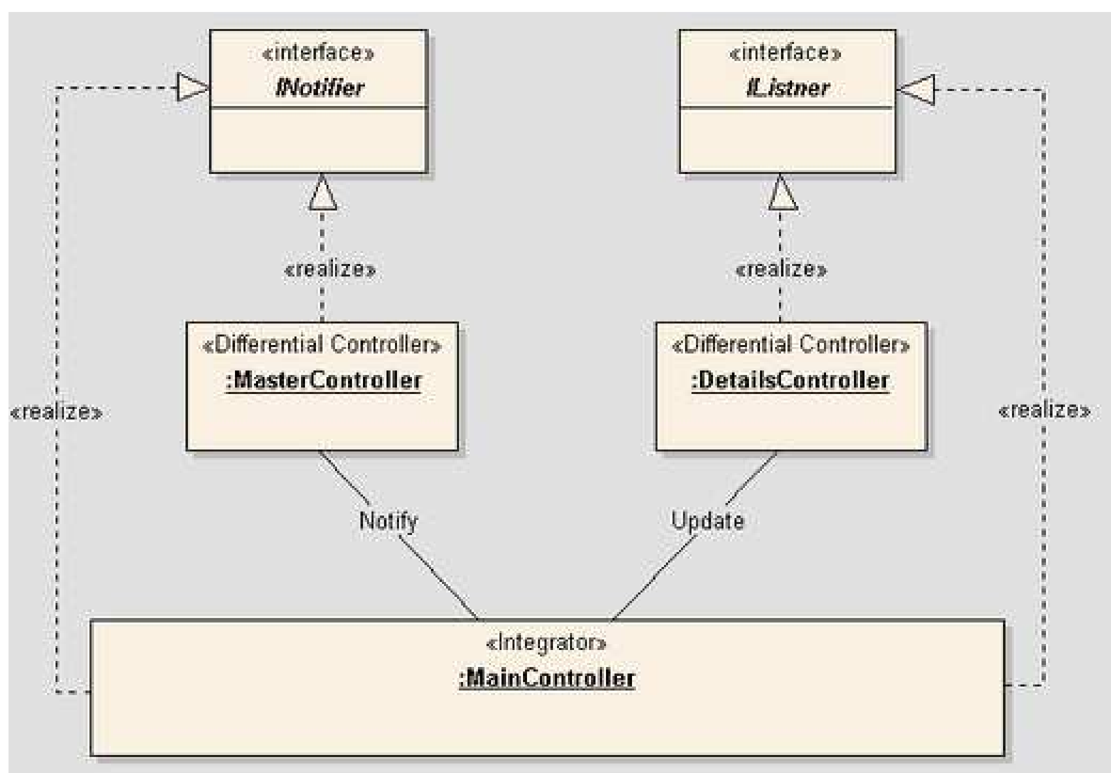
Rysunek 27: Diagram przypadków użycia



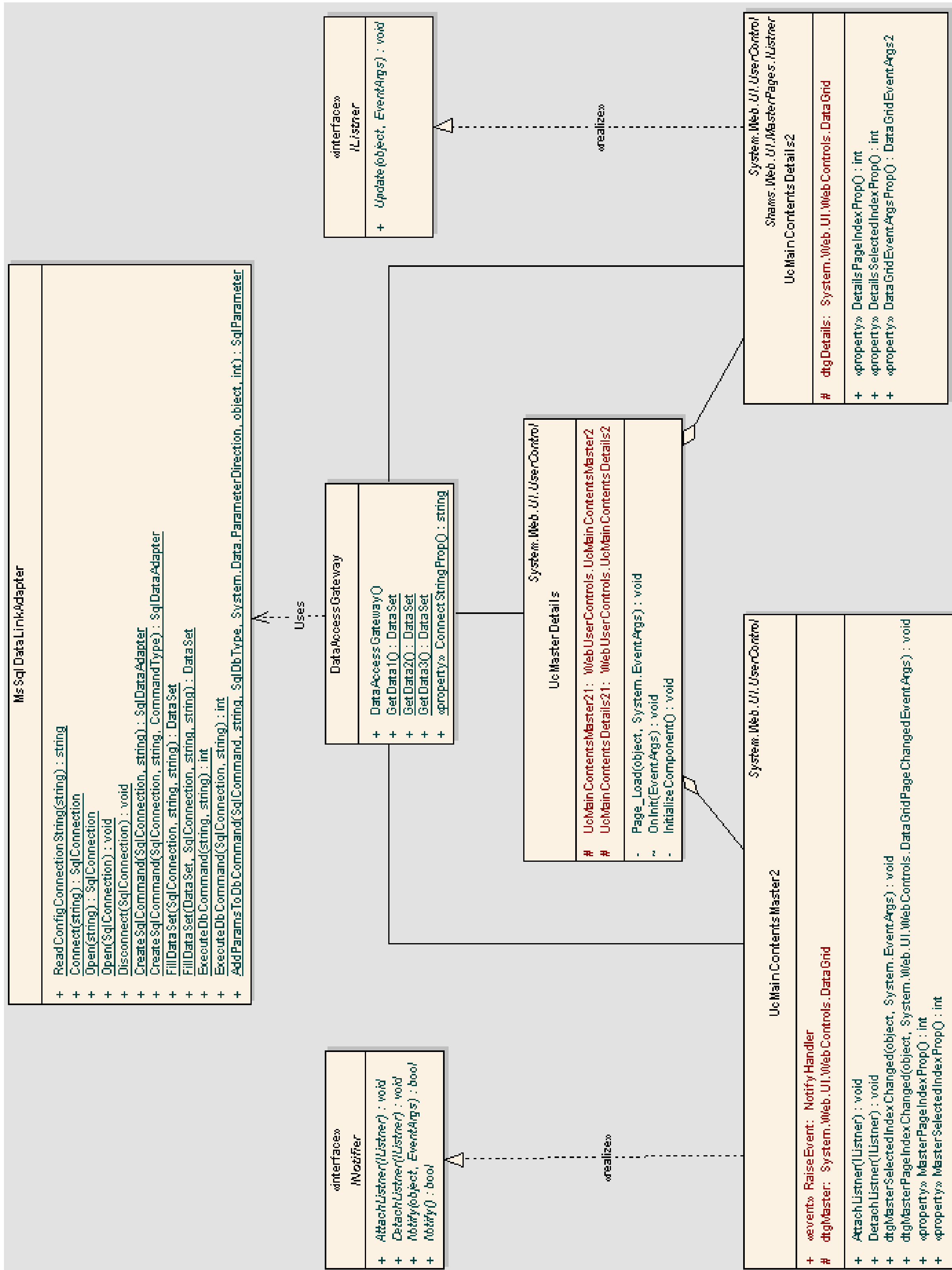
Rysunek 28: Przykładowy layout



Rysunek 29: Robustness Analysis diagram



Rysunek 30: Diagram interakcji



Rysunek 31: Diagram klas

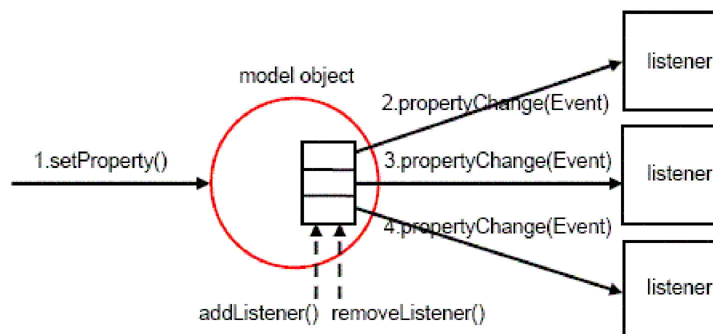
5.2.3. GEF i EMF

GEF Graphical Editing Framework to narzędzie służące do budowy edytorów graficznych na podstawie istniejącego modelu aplikacji. Składa się on z wtyczek udostępniających podstawowe funkcje do zarządzania wyglądem i ustawieniami komponentów oraz wyświetlania grafiki 2D. GEF korzysta z wzorca *MVC* ułatwiającego budowę aplikacji typu GUI, patrz [27].

Model, patrz Rys. 32, przechowuje dane aplikacji, jest niezależny od widoku oraz musi implementować mechanizm powiadamiania. Istnieją dwie strategie powiadamiania:

- scentralizowana - istnieje jeden obiekt powiadamiający o zmianach,
- rozproszona - każdy obiekt modelu powiadamia o swoich zmianach.

Model można podzielić na Model Dziedziny reprezentujący część biznesową oraz Model Prezentacji reprezentujący sposób wyświetlania (klasa modelu musi implementować interfejs `IPropertySource`, aby możliwe było modyfikowanie właściwości modelu przy wykorzystaniu Properties Eclipsa). GEF stosuje Model Pasywny, opisany z Rozdziale 2.2.1, oraz korzysta ze wzorca *Obserwator*, patrz Rozdział 3.1, do powiadamiania kontrolerów o zmianach.

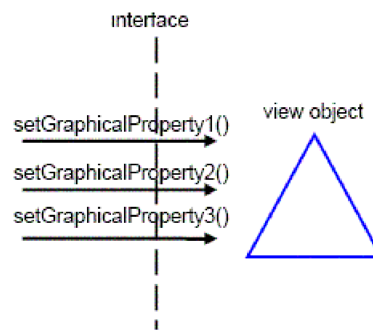


Rysunek 32: Model

Model można oprzeć na Plain Old Java Objects - POJO. Alternatywą jest użycie *EMF Eclipse Modeling Framework* - narzędzia do modelowania i generowania kodu na podstawie modelu danych. EMF potrafi wygenerować podstawowe klasy Javy oraz inne klasy pomocnicze, pozwalające na przykład na edycję modelu. Modele mogą być zarówno zapisane w plikach XML, jak i zawarte w klasach Javy oznaczonych adnotacjami czy też importowane z innych narzędzi (np. Rational Rose). EMF dostarcza adapterów nasłuchujących zmian w modelu, co jest jednym z wymagań w przypadku budowy modelu w GEF. EMF .edit framework wspiera budowę adapterów oraz edycję właściwości modelu. Integracja EMF .edit oraz GEF może być kłopotliwa, ponieważ frameworki korzystają z różnych interfejsów komend.

Komponenty widoku, patrz Rys. 33, są definiowane za pomocą pluginu Draw2D dostarczającego predefiniowanych kształtów. Widok nie posiada odwołań do modelu. Każdy z obiektów reprezentujących elementy widoku dziedziczy po klasie `Figure` oraz implementuje interfejs `IFigure`. Możliwe jest korzystanie z podstawowych kształtów, budowanie komponentów zbudowanych z elementów predefiniowanych oraz tworzenie własnych komponentów. GEF dostarcza także różnych managerów rozkładu.

Kontroler stanowi most pomiędzy widokiem a modelem. Według konwencji GEF kontrolery to obiekty dziedziczące po klasie `EditPart`, odpowiedzialne za mapowanie modelu i widoku,



Rysunek 33: Widok

modyfikowanie modelu oraz obserwowanie modelu (wzorzec Obserwator) i aktualizowanie widoku w odpowiedzi na zmiany w modelu. Klasy rozszerzające `EditPart` muszą implementować następujące metody:

- `createFigure()` - zwraca komponent graficzny,
- `getModelChildren()` - implementowana, gdy istnieje relacja dziedziczenia,
- `getContentPane()` - zwraca kontener,
- `getModelSourceConnections()`, `getModelTargetConnections()`, `getSourceConnectionAnchor()` - implementowane przez obiekty reprezentujące węzły,
- `createEditPolicies()` - zmiany w modelu w odpowiedzi na akcje użytkownika,
- `activate()`,`deactivate()` - dodawanie i usuwanie listnerów,
- `refreshVisuals()`, `refreshChildren()`, `refreshSourceConnections()`, `refreshTargetConnections()` - odświeżanie widoku.

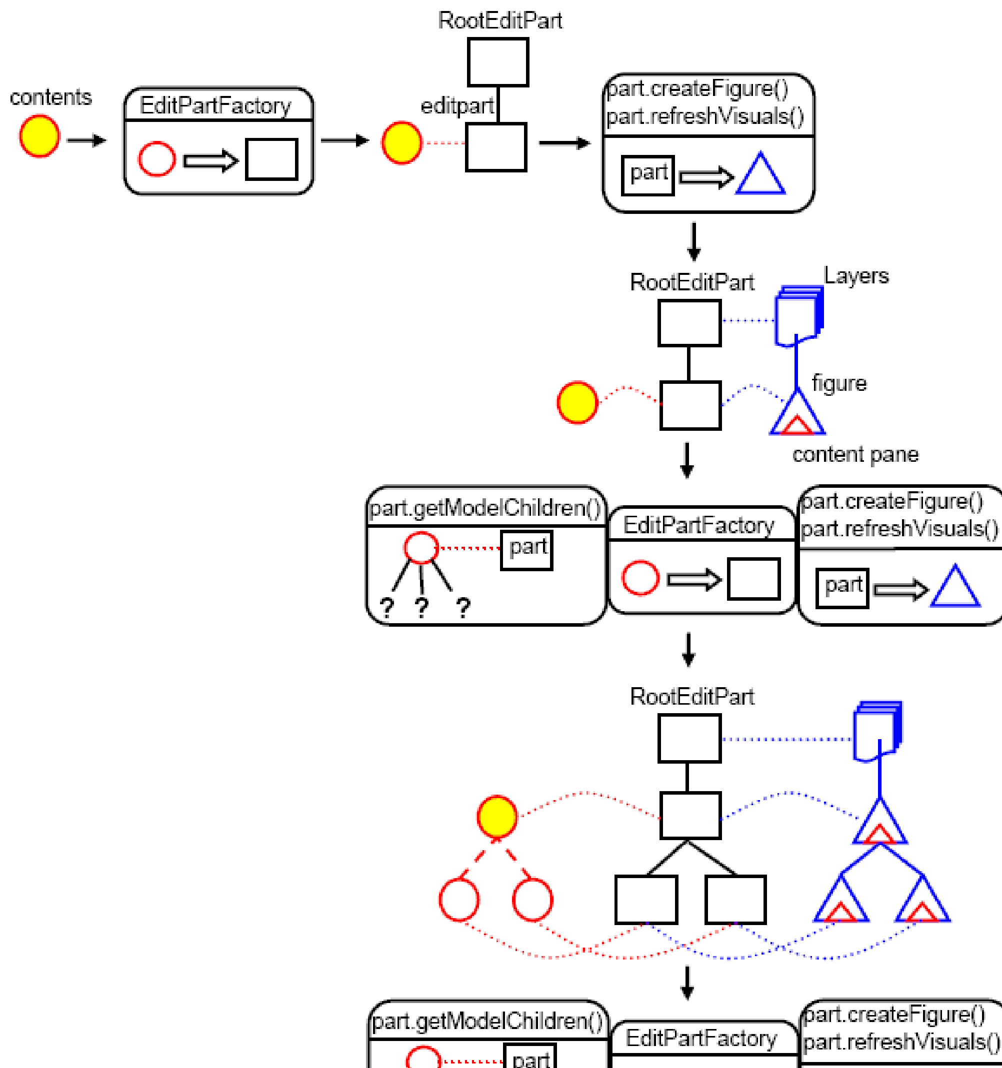
Tworzenie komponentów widoku : współpracują tu następujące obiekty: `EditPartFactory` (tworzy obiekty `EditPart` odpowiadające obiektom modelu), `EditPartViewer` (instalowanie widoków), komponenty Draw2D. Proces budowy widoku, patrz Rys. 34, ma charakter rekurencyjny. Model jest przekazywany do `EditPartViewer`, `EditPartFactory` tworzy główny `EditPart` , metody `createFigure()` oraz `refreshVisuals()` tworzą komponenty graficzne, a `getModelChildren()` pobiera obiekty modelu znajdujące się niżej w hierarchii dziedziczenia. Dla każdego z obiektów modelu `EditPartFactory` tworzy obiekt `EditPart` . Proces powtarza się od początku.

Odświeżanie komponentów widoku : model powiadamia nasłuchujące go `EditParts` o zmianach, widoki odświeżane są poprzez metody `refreshXXX()`, patrz Rys. 35.

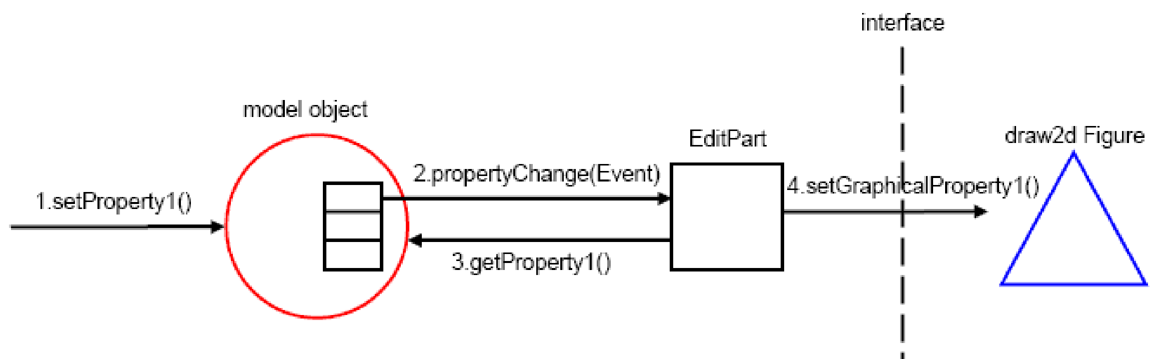
Modyfikowanie modelu przedstawione jest na Rys. 36:

1 Użytkownik wykonuje akcję,

2 Akcje użytkownika są przechwytywane przez `EditPartViewer`,



Rysunek 34: Tworzenie komponentów widoku



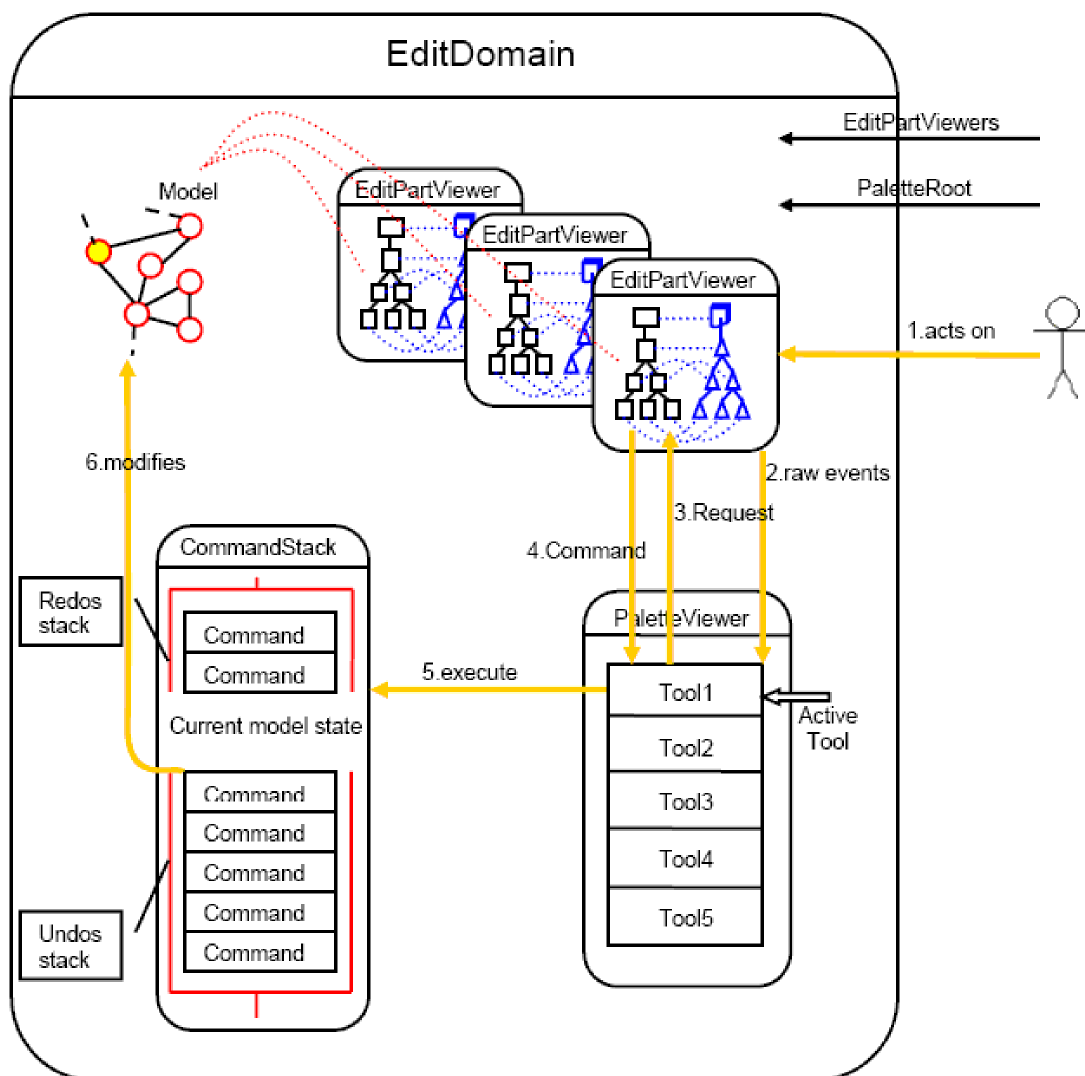
Rysunek 35: Odświeżanie komponentów widoku

3 Tworzone są obiekty `Request`, zawierające informacje jakie akcje mają być wykonane,żądanie są przesyłane do obiektu `EditPart`,

4 `EditPart` zwraca odpowiedni obiekt `Command`,

5 `Command` przekazywane jest na stos `CommandStack`,

6 Metody komendy modyfikują model, widoki są odświeżane poprzez mechanizm powiadamiania.



Rysunek 36: Odświeżanie komponentów widoku

6. Podsumowanie

Wzorzec *MVC Model View Controller* jest jednym z najczęściej opisywanych wzorców projektowych. Od czasu swego powstania, tj. późnych lat 70. odegrał on znaczącą rolę w sposobie

projektowania interfejsu użytkownika. We wzorcu MVC występuje wyraźny podział na trzy role:

- model to obiekt reprezentujący informacje dotyczące dziedziny,
- widok reprezentuje prezentację modelu w interfejsie użytkownika,
- kontroler pobiera dane wprowadzane przez użytkownika, modyfikuje model i wymusza odpowiednią aktualizację widoku.

Istnieją dwie kluczowe granice pomiędzy elementami wzorca *MVC*. **Rozdzielenie prezentacji od modelu** jest jedną z najważniejszych zasad projektowania oprogramowania. Jest ono istotne ze względu na zupełnie inne przeznaczenie tych komponentów. Widok to mechanizmy interfejsu użytkownika natomiast model reprezentuje funkcje biznesowe. Rozdzielenie prezentacji od modelu pozwala na stworzenie wielu różnych prezentacji korzystając z tego samego modelu. Kolejną zaletą jest fakt, że możliwe jest testowanie kodu obsługującego logikę biznesową w sposób niezależny, co jest znacznie prostsze. Prezentacja jest uzależniona od modelu, natomiast model jest niezależny od prezentacji. Dzięki temu aplikacja może być w prosty sposób rozszerzana. Model może być Modelem Pasywnym(kontroler uaktualnia widoki) bądź Aktywnym(model sam powiadamia nasłuchujące obiekty o zmianach - wzorzec *Obserwator*).

Rozdzielenie widoku od kontrolera ma drugorzędne znaczenie. W aplikacjach typu GUI każdy widok posiada tylko jeden kontroler, stąd rozdzielenie to nie jest zazwyczaj realizowane. Odseparowanie widoku od kontrolera odgrywa znaczącą rolę w aplikacjach webowych, gdzie interfejsem użytkownika są strony WWW. Funkcje kontrolera są różne w aplikacjach webowych:

- *Application Controller* - kontroler jest zlokalizowany pomiędzy widokiem a modelem,
- *Page Controller* - każdy widok posiada swój własny kontroler,
- *Front Controller* - kontroler konsoliduje obsługę żądań poprzez skierowanie ich do jednego obiektu.

Podsumowując pracę należałoby wymienić wady oraz zalety stosowania architektury *MVC* w aplikacji. Twórcy aplikacji powinni bowiem w pełni rozumieć zalety i wady wynikające ze stosowania opisanych zaleceń projektowych.
Do zalet *MVC* należy:

- wyraźnie rozdzielenie elementów o różnym przeznaczeniu,
- możliwość specjalizacji i rozwijania jednego z komponentów MVC,
- równoległa praca zespołów,
- łatwa zamiana interfejsu użytkownika w istniejącej aplikacji,
- łatwość testowania logiki biznesowej,
- dostępność bibliotek i środowisk wspomagających budowę aplikacji w oparciu o architekturę *MVC*.

Wady *MVC* są następujące:

- większa złożoność projektu,
- zwielokrotniona liczba komponentów, które muszą być zaimplementowane,
- silne sprzężenie na poziomie interfejsów - zmiany interfejsów w modelu pociągają za sobą konieczność modyfikacji widoku i kontrolera.

Przedstawiona w pracy koncepcja wzorca *MVC Model View Controller* jest ze względu na swoje zalety często wykorzystywana w tworzeniu aplikacji - zarówno typu GUI jak i aplikacji webowych. O popularności wzorca świadczy różnorodność frameworków wykorzystujących *MVC. Model View Controller* ułatwia koordynację prac projektowych oraz pielęgnację wytworzonego oprogramowania.

Literatura

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe: elementy oprogramowania obiektowego wielokrotnego użytku*, tłum. Janusz Jabłonowski, WNT, Warszawa 2005.

[2] James W. Cooper, *Java Wzorce projektowe*, tłum. Piotr Badarycz, Wydawnictwo HELION, Gliwice 2000.

[3] Martin Fowler, David Rice, Matthiew Foemmel, Edward Hieatt, Robert Mee, Rendy Stafford, *Architektura systemów zarządzania przedsiębiorstwem Wzorce projektowe*, tłum. Piotr Rajca, Paweł Koronkiewicz, Wydawnictwo HELION, Gliwice 2005.

[4] Jeff Moore, *Model View Controller*, `http://www.phpwact.org/pattern/model_view_controller`, 2006.12.09.

[5] John Deacon, *Model-View-Controller (MVC) Architecture*, `http://www.jdl.co.uk/briefings/MVC.pdf`, 2005 August.

[6] Martin Folwer, *Development of Further Patterns of Enterprise Application Architecture*, `http://martinfowler.com/eaaDev/`.

[7] Core J2EE Pattern Catalog, `http://www.corej2eepatterns.com/Patterns2ndEd/index.htm`.

[8] Microsoft patterns & practices, `http://msdn2.microsoft.com/en-us/library/ms998572.aspx`.

[9] Design Patterns, `http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/default.htm`.

[10] Patterns Catalog, `http://hillside.net/patterns/onlinepatterncatalog.htm`.

[11] Steve Burbeck, *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*, `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`.

[12] Glenn E. Krasner, Stephen T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*, ParcPlace Systems, 1988.

[13] Understanding Rails MVC, http://wiki.rubyonrails.com/rails/pages/UnderstandingRailsMVC.

[14] Framework Django, http://www.python.rk.edu.pl/w/p/djangoindex/.

[15] Framework Pylons do tworzenia aplikacji internetowych, http://www.python.rk.edu.pl/w/p/pylonsindex/.

[16] CakePHP, http://www.cakephp.org/.

[17] Symfony, http://www.symfony-project.com/.

[18] Implementing Model-View-Controller in ASP.NET, http://msdn2.microsoft.com/en-us/library/ms998540.aspx.

[19] *Designing Enterprise Applications with the J2EE Platform, Second Edition*, http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch5.html.

[20] Krzysztof Barteczko, *Java Od podstaw do technologii*, Wydawnictwo MIKOM, Warszawa 2004.

[21] Rod Johnson, *Introduction to the Spring Framework*, http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework.

[22] The Spring Framework - Reference Documentation, http://www.springframework.org/docs/reference/.

[23] Martin Folwer, *GUI Architectures*, http://www.martinfowler.com/eaaDev/uiArchs.html, 2006.06.18.

[24] David J. Anderson, *Using MVC Pattern in Web Interactions*, http://www.uidesign.net/Articles/Papers/UsingMVCPatterninWebInter.html, 2000.06.01.

[25] Jean-Michel Garnier, *Struts 1.1 Controller UML diagrams*, http://rollerjm.free.fr/pro/Struts11.html, 2003.12.19.

[26] Shams Mukhtar, *Applying Robustness Analysis on the Model-View-Controller (MVC) Architecture in ASP.NET Framework, using UML*, http://www.codeproject.com/aspnet/ModelViewController.asp, 2004.08.23.

[27] GefDescription, http://eclipsewiki.editme.com/GefDescription, 2006.06.06.