Warstwa prezentacji

wg. D.Alur, J.Crupi, D. Malks, Core J2EE. Wzorce projektowe.

1. Definicja warstwy prezentacji - pięciowarstwowy model logicznego rozdzielania zadań
2. Podstawowe przypadki - analiza podstawowych przypadków

Pięciowarstwowy model logicznego rozdzielania zadań (wg. D.Alur, J.Crupi, D. Malks, Core J2EE. Wzorce projektowe.)

Problem 1 – Należy przechwycić i modyfikować żądanie i odpowiedź przed i po właściwym przetwarzaniu

Uwagi:

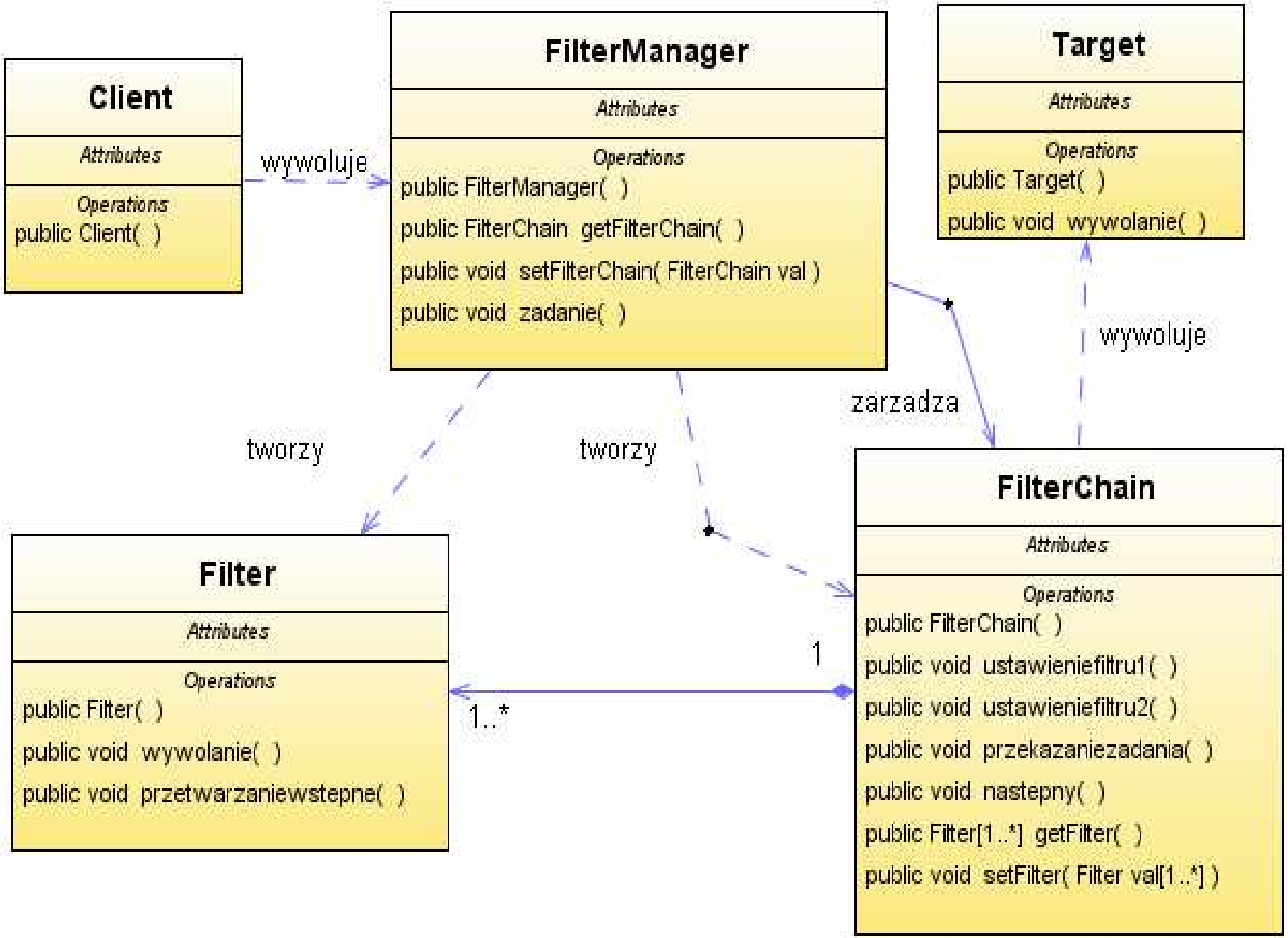
1. Czy z klientem związana jest poprawna sesja?
2. Czy ścieżka narusza jakieś ograniczenia?
3. Czy jest obsługiwany dany rodzaj przeglądarki internetowej?
4. Jakiego kodowania używa klient do wysyłania danych?
5. Czy strumień danych jest zaszyfrowany lub skompresowany?

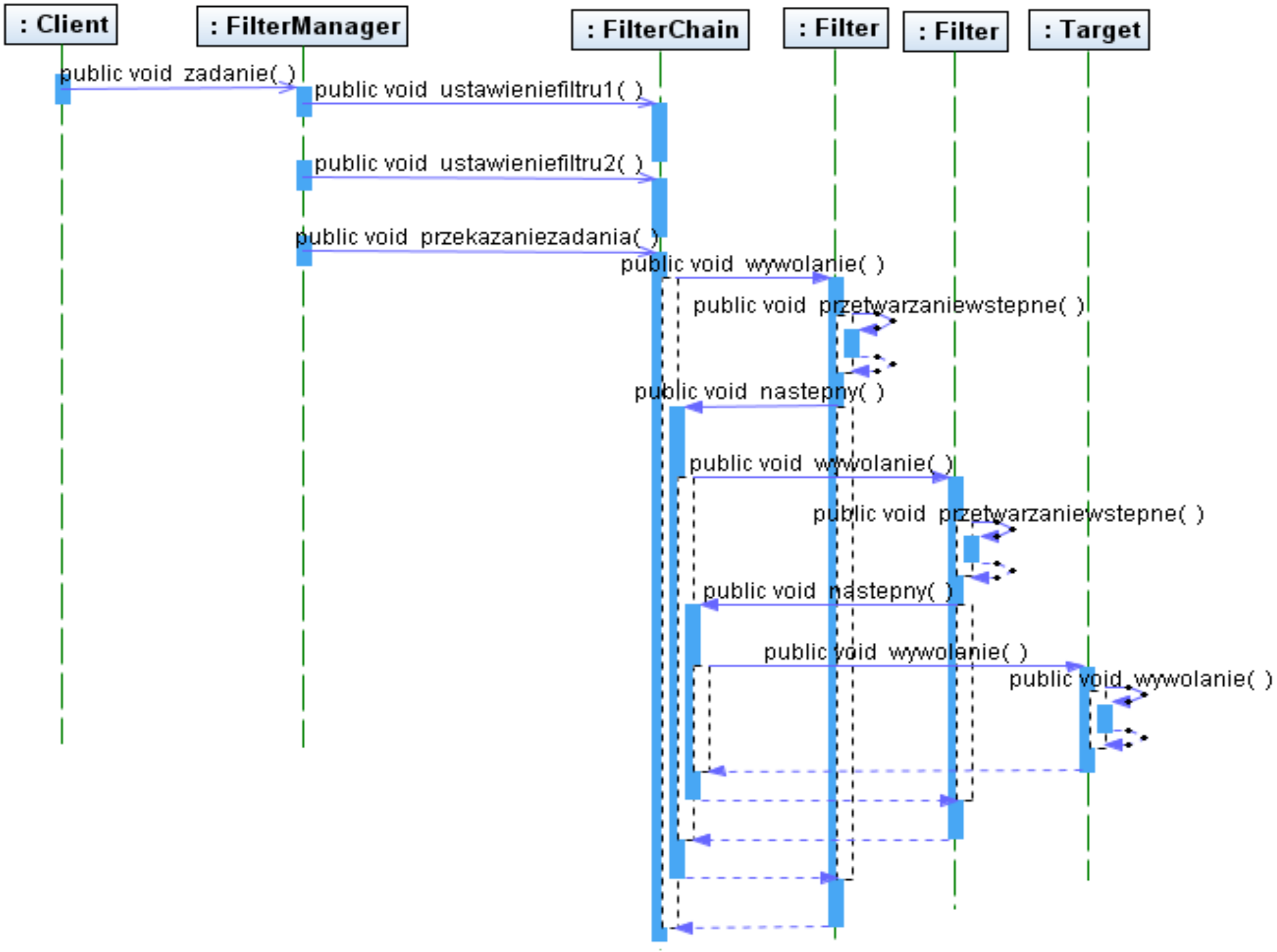
Wymagania:

1. Należy **scentralizować** i **wspólnie** przetwarzać wszystkie żądania, na przykład sprawdzenie kodowania, zapamiętywania informacji o każdym żądaniu, wprowadzenia kompresji danych dla odpowiedzi wysyłanej klientowi.
2. Nie należy mocno wiązać przetwarzania wstępnego i końcowego z głównym kodem przetwarzania żądań, aby ułatwić dodawanie i usuwanie, na przykład nowych sposobów kompresji.
3. Należy zastosować **niezależne komponenty** służące do przetwarzania wstępnego i końcowego w celu zwiększenia możliwości ich wielokrotnego wykorzystania.

Wzorzec **Intercepting Filter** jako filtr przetwarzania wstępnego i końcowego żądań

Jest to zarządca filtrów, który łączy powiązane filtry podstawowe w łańcuch, przekazując do nich sterowanie. Można tworzyć łańcuch dowolnych filtrów bez zmiany istniejącego kodu.


Client
Attributes
Operations
public Client( )
wywoluje
FilterManager
Attributes
Operations
public FilterManager( )
public FilterChain getFilterChain( )
public void setFilterChain(FilterChain val)
public void zadanie( )
Target
Attributes
Operations
public Target( )
public void wywolanie( )
wywoluje
zarzadza
tworzy
tworzy
Filter
Attributes
Operations
public Filter( )
public void wywolanie( )
public void przetwarzaniewstepne( )
1
1..*
FilterChain
Attributes
Operations
public FilterChain( )
public void ustawieniefiltru1( )
public void ustawieniefiltru2( )
public void przekazaniezadania( )
public void nastepny( )
public Filter[1..*] getFilter( )
public void setFilter(Filter val[1..*])


: Client
: FilterManager
: FilterChain
: Filter
: Filter
: Target
public void zadanie( )
public void ustawieniefiltru1( )
public void ustawieniefiltru2( )
public void przekazaniezadania( )
public void wywolanie( )
public void przetwarzaniewstepne( )
public void nastepny( )
public void wywolanie( )
public void przetwarzaniewstepne( )
public void nastepny( )
public void wywolanie( )
public void wywolanie( )

Problem 2 – Potrzebny jest scentralizowany punkt dostępowy dla obsługi żądań w warstwie prezentacji

Uwagi:

Jeśli brakuje centralizacji punktu dostępowego dla obsługi żądań – wtedy kod sterujący powtarza się w wielu miejscach (np. w widokach). Takie rozwiązanie nie jest modułowe i elastyczne i utrudnia pielęgnację kodu.

Wymagania:

1. Należy uniknąć powielenia logiki sterującej.
2. Należy stosować wspólną logikę dla wielu rozwiązań
3. Należy oddzielić logikę przetwarzania od widoku.
4. Należy scentralizować i kontrolować wszystkie punkty dostępu do systemu

Definicje:
Przetwarzanie żądania:

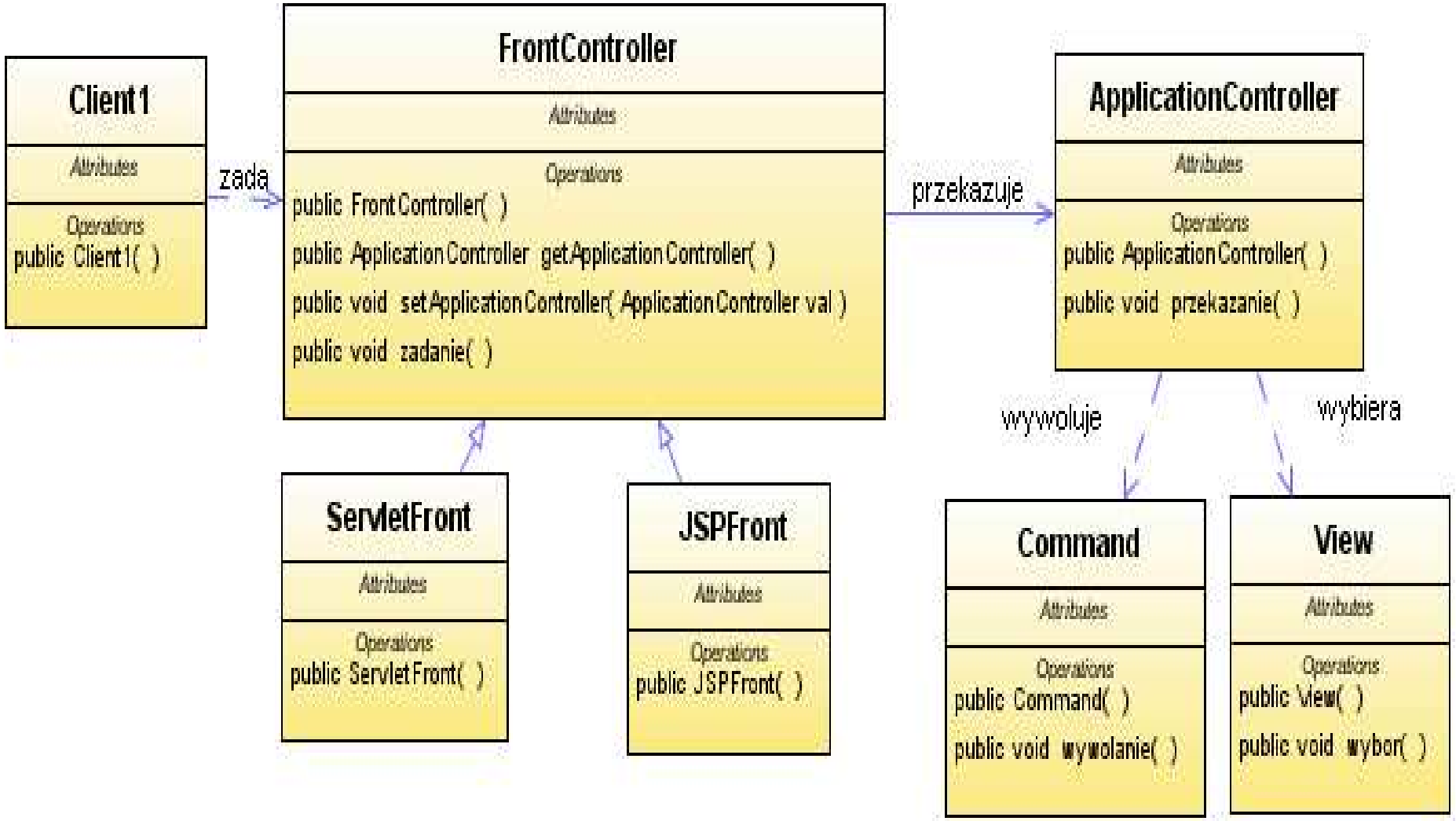
- *Obsługa żądania*
  - obsługa protokołu i operacje na kontekście
  - nawigacja i wybór ścieżki
  - przetwarzanie główne (działania na serwerze)
  - przekazanie sterowania (dispatch)
- *Przetworzenia widoku* po przekazaniu sterowania z obsługi żądania do komponentów przetwarzania widoku

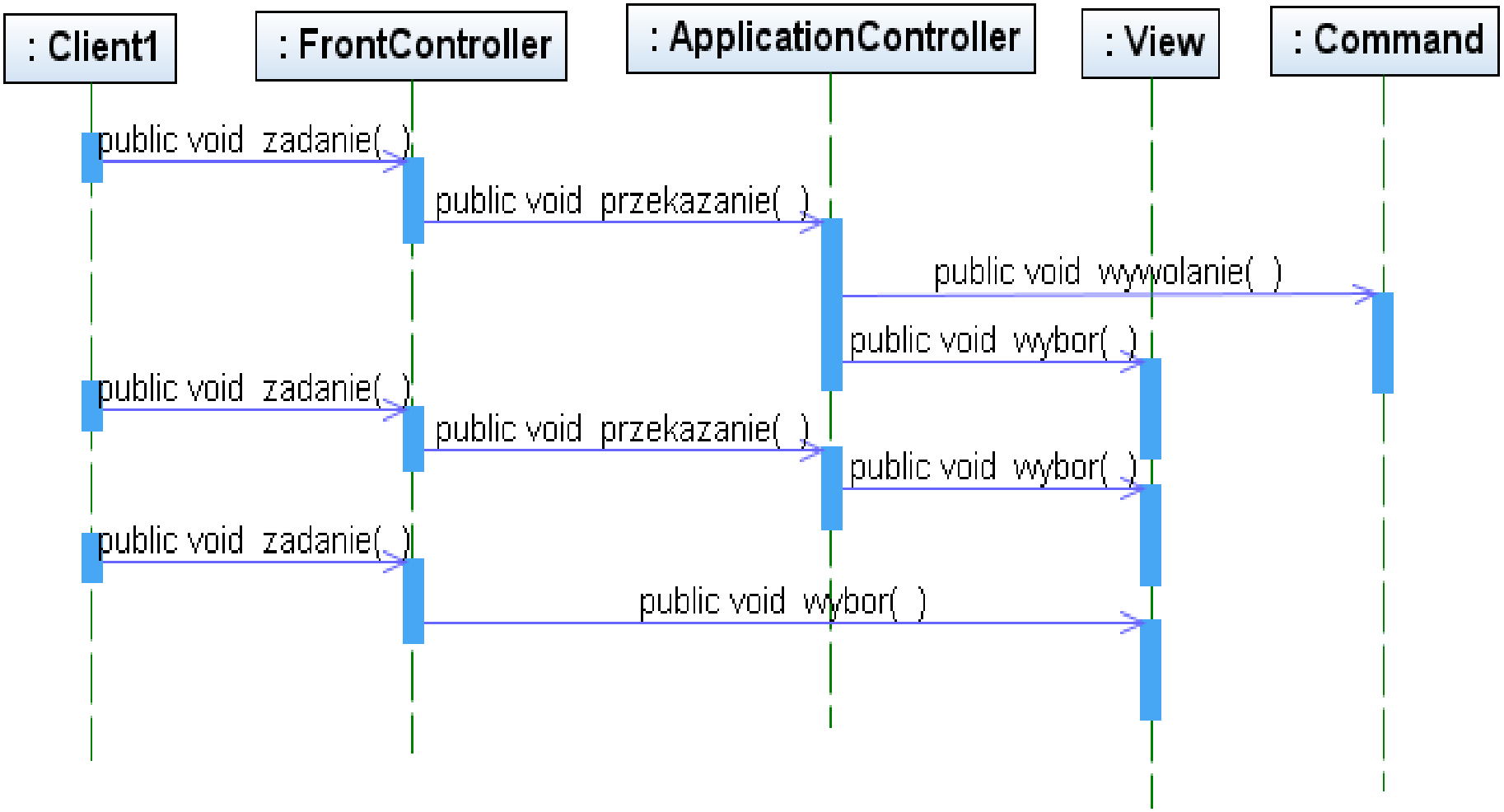
Wzorzec **Front Controller,** jako początkowe miejsce obsługi wszystkich żądań:

- centralizuje logikę sterującą, która w innej sytuacji byłaby z pewnością powielona w różnych miejscach,
- zajmuje się podstawowymi operacjami związanymi z obsługą żądań.

Wzorzec **Application Controller** jest wykorzystywany przez wzorzec Front Controller do:

- Zarządzania akcjami - dotyczy wyszukiwania odpowiednich usług i przekazywania do nich sterowania w celu obsłużenia konkretnych żądań klientów (**Command**)
- Zarządzania widokami zwracanymi klientowi(**View**)- dotyczy wyszukiwania i odwoływania się do odpowiedniego widoku. Choć zadanie to może stanowić część odpowiedzialności wzorca **Front Controller**, zastosowanie osobnej klasy jako części wzorca **Application Controller** zwiększa modułowość i możliwość wielokrotnego użycia oraz ułatwia pielęgnację kodu.


Client1
Attributes
Operations
public Client1( )
zada
FrontController
Attributes
Operations
public FrontController( )
public ApplicationController getApplicationController( )
public void setApplicationController(ApplicationController val)
public void zadanie( )
przekazuje
ApplicationController
Attributes
Operations
public ApplicationController( )
public void przekazanie( )
wywołuje
wybiera
ServletFront
Attributes
Operations
public ServletFront( )
JSPFront
Attributes
Operations
public JSPFront( )
Command
Attributes
Operations
public Command( )
public void wywołanie( )
View
Attributes
Operations
public View( )
public void wybor( )


: Client1
: FrontController
: ApplicationController
: View
: Command
public void zadanie( )
public void przekazanie( )
public void wywolanie( )
public void wybor( )
public void zadanie( )
public void przekazanie( )
public void wybor( )
public void zadanie( )
public void wybor( )

Problem 3 – Należy uniknąć korzystania z informacji specyficznych dla protokołu poza właściwym mu kontekstem

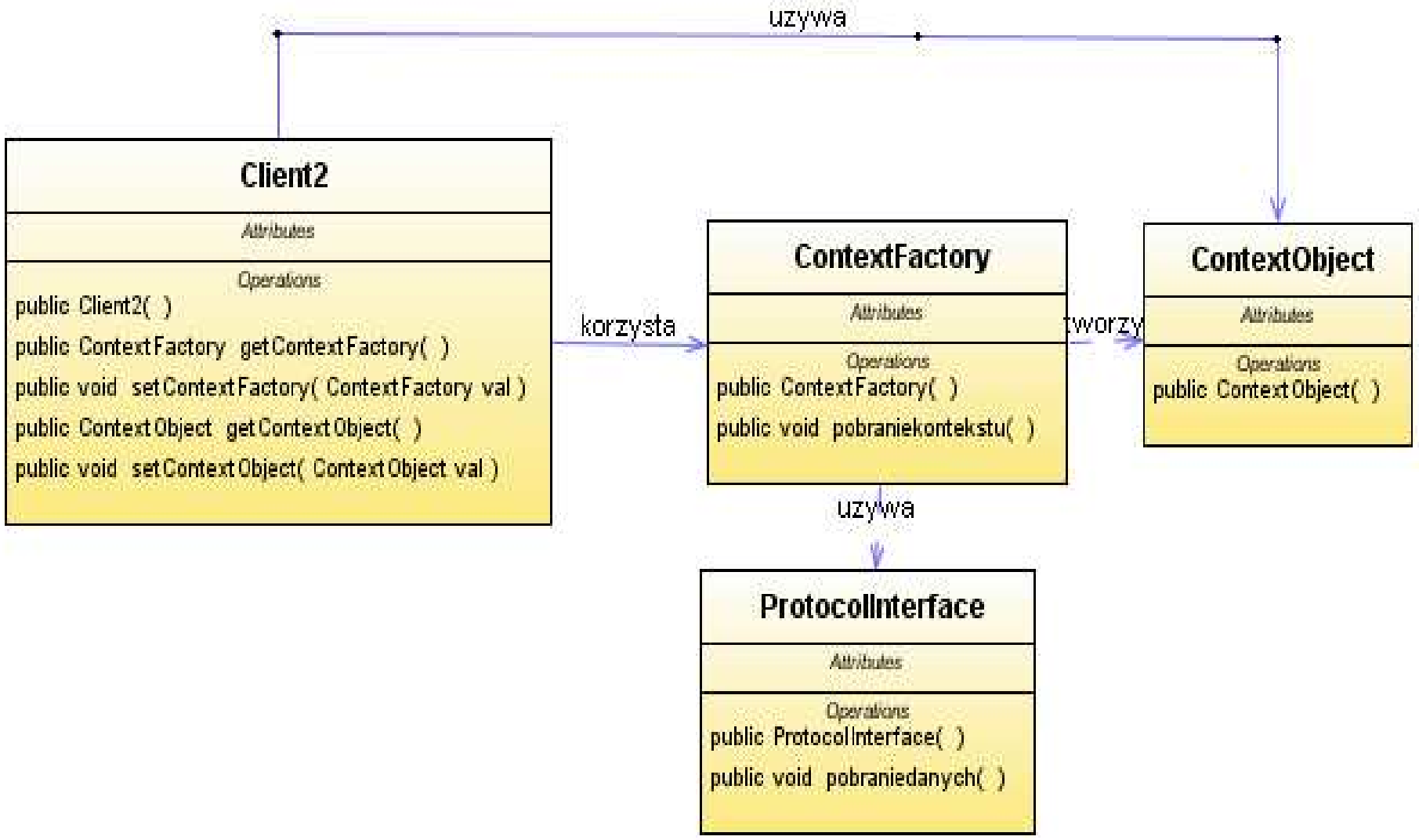
Uwagi:

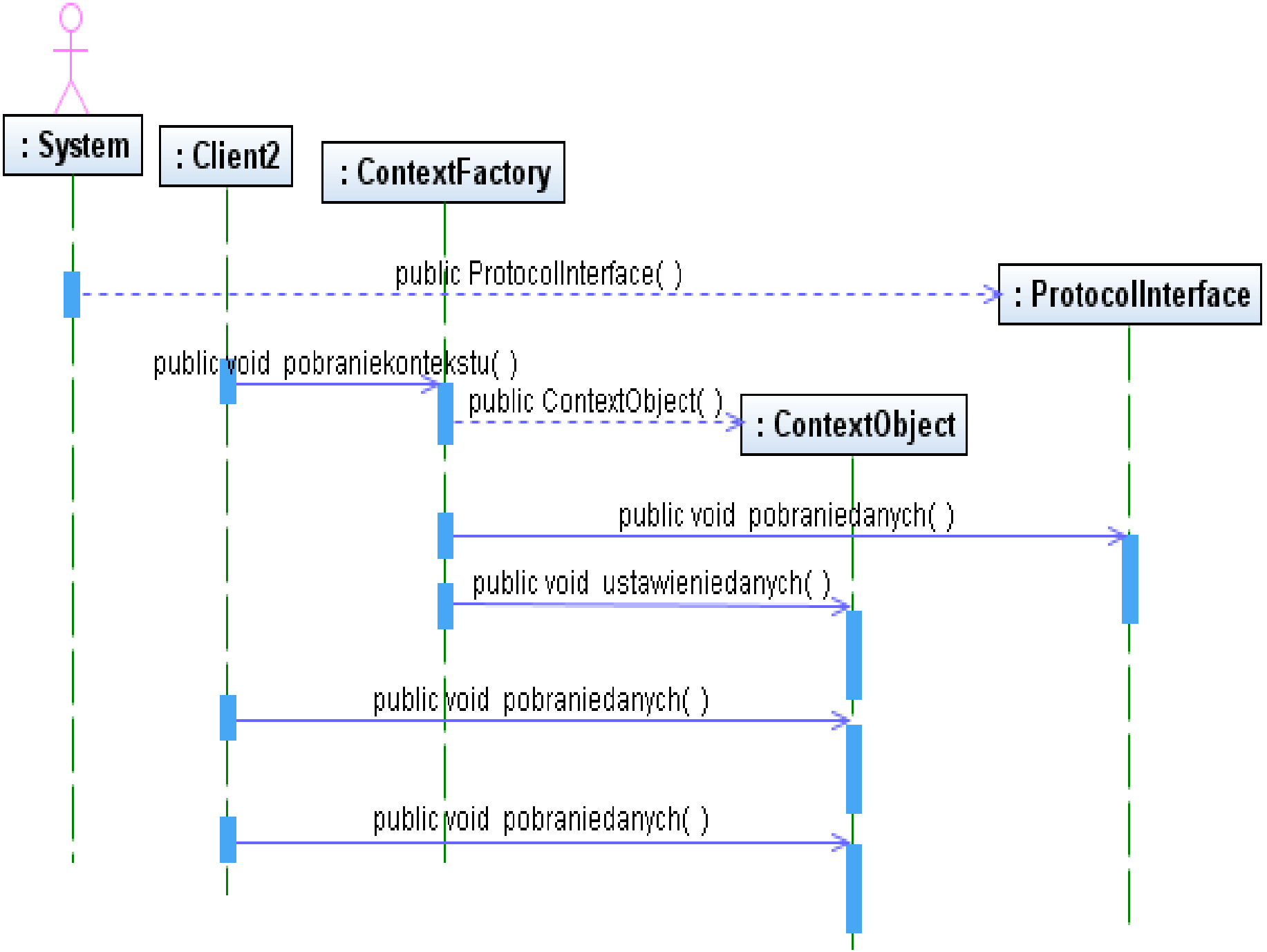
1. Aplikacja korzysta z pewnych informacji systemowych, takich jak żądania klienta, dane konfiguracyjne i dane związane z bezpieczeństwem w trakcie cyklu życia obiektów żądania i odpowiedzi. Dane te czerpie ona z odpowiedniego kontekstu.
2. Informacje nie należące do kontekstu komponentu nie powinny być mu udostępniane, ponieważ zmniejsza się jego użyteczność i elastyczność.

Wymagania:

1. Komponenty i usługi wymagają dostępu do informacji systemowych.
2. Należy oddzielić komponenty aplikacji i usługi od szczegółów protokołu.
3. W danym kontekście należy ujawnić tylko potrzebne elementy interfejsu.

Wzorzec **Context Object** umożliwia hermetyzację stanu w sposób niezależny od protokołu i udostępnia go kolejnym elementom aplikacji.




: System
: Client2
: ContextFactory
public ProtocolInterface( )
: ProtocolInterface
public void pobraniekontekstu( )
public ContextObject( )
: ContextObject
public void pobraniedanych( )
public void ustawieniedanych( )
public void pobraniedanych( )
public void pobraniedanych( )

Problem 4 – należy scentralizować i uzyskać modułową strukturę zarządzania akcjami aplikacji i widokami

Uwagi:

1. Wewnątrz warstwy prezentacji po otrzymaniu żądania podejmuje się dwie decyzje:
   - Otrzymane żądanie musi zostać przekazane do przetworzenia przez odpowiednią akcję realizującą żądaną usługę – jest to zarządzanie akcjami
   - Lokalizowany i wykorzystywany jest odpowiedni widok – jest to zarządzanie widokami
2. Zarządzanie akcjami i widokami można zawrzeć w różnych częściach aplikacji. Umieszczenie tego zachowania w implementacji **Front Controller** centralizuje tę funkcjonalność, jednak wraz z rozwojem aplikacji warto zastosować osobną grupę klas zwiększając modułowość, rozszerzalność i elastyczność rozwiązania.

Wymagania:

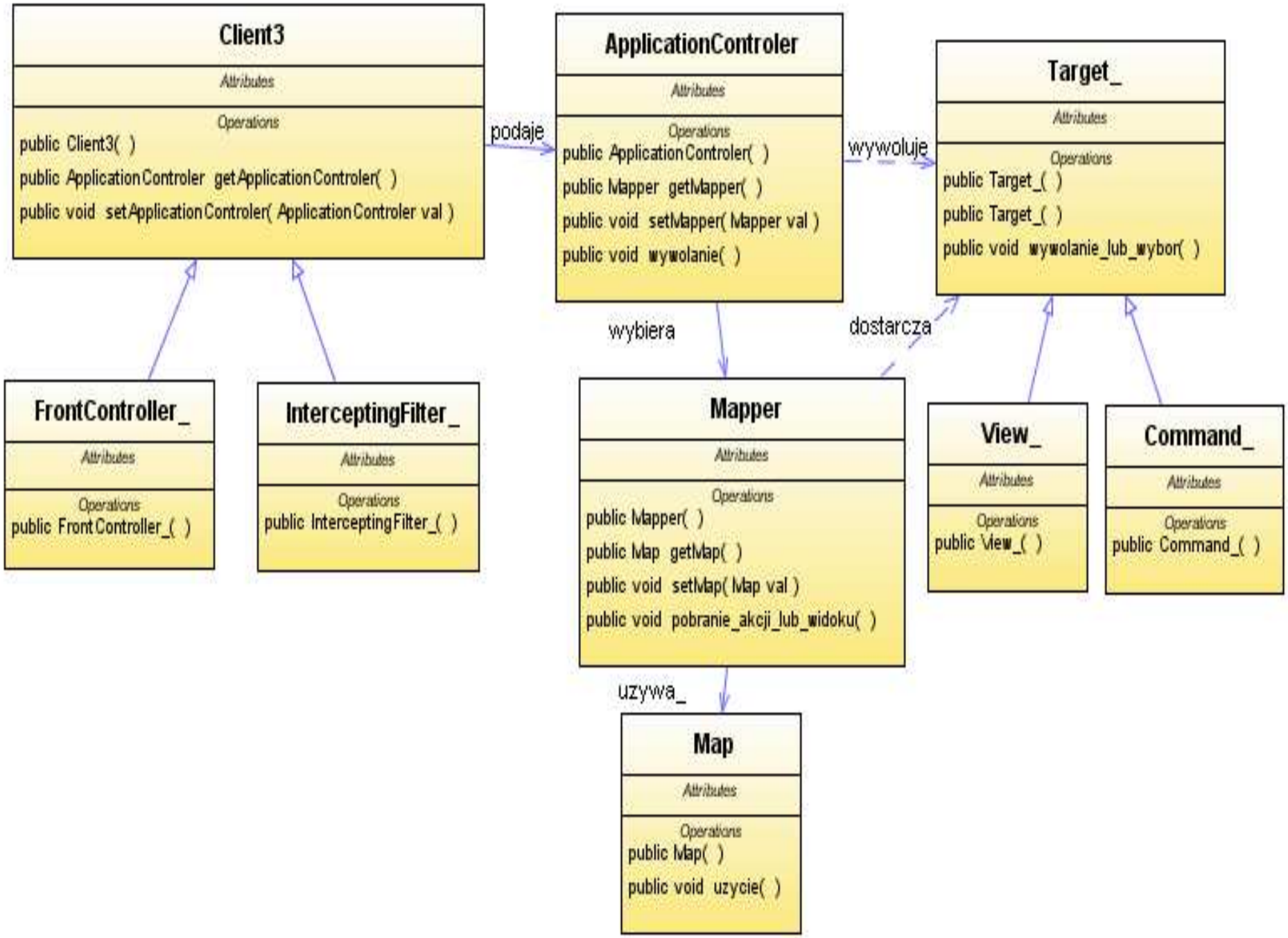
1. Należy wielokrotnie korzystać z kodu zarządzania akcjami aplikacji i widokami
2. Należy poprawić rozszerzalność obsługi żądań, na przykład przez możliwość stopniowego dodawania implementacji kolejnych przypadków użycia.
3. Należy zwiększyć modułowość kodu, by ułatwić rozszerzanie funkcjonalności aplikacji i ułatwić testowanie kodu obsługi poszczególnych żądań niezależnie od środowiska udostępnianego przez kontener web, czy serwer aplikacji.

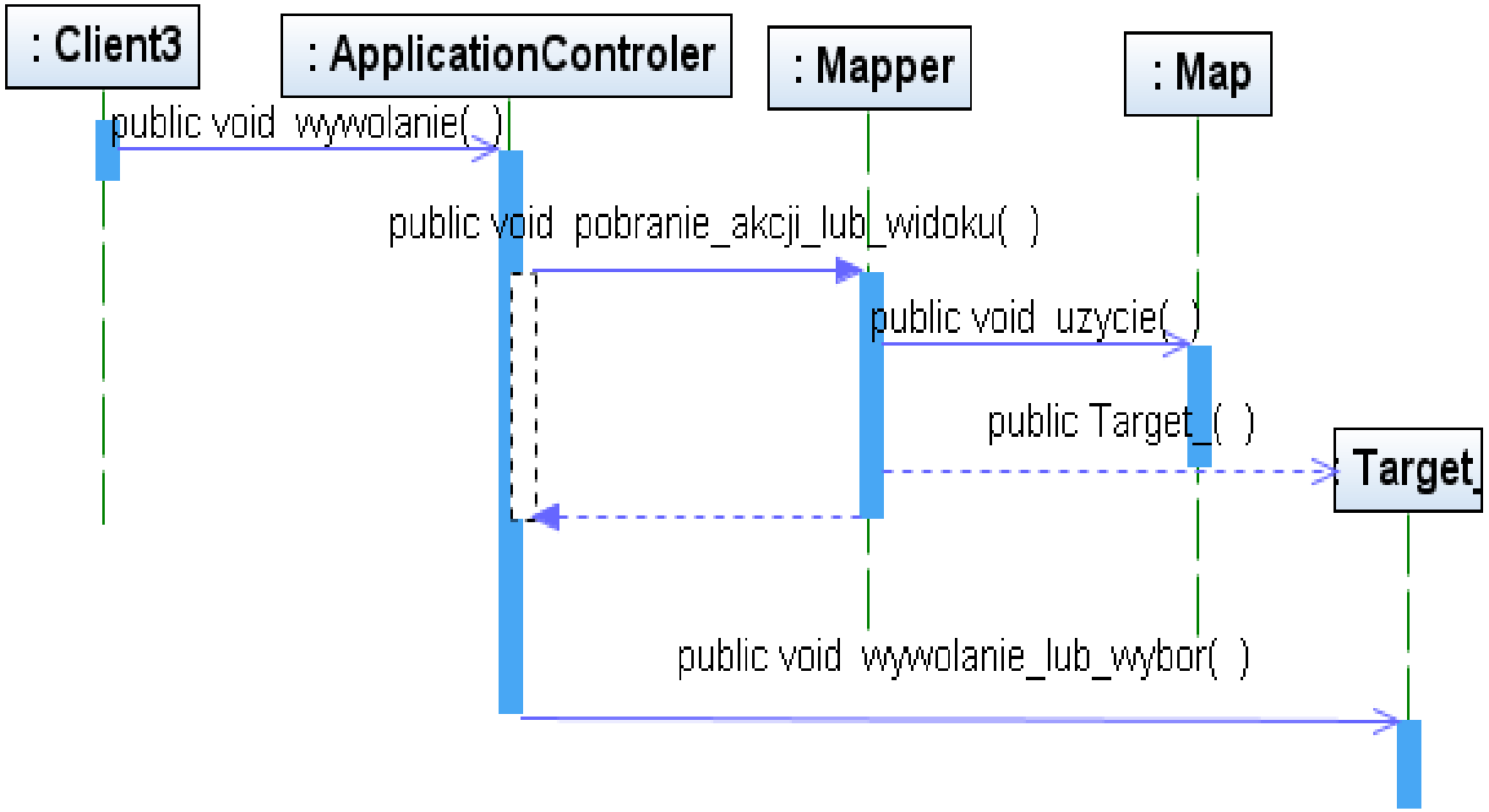
Wzorzec **Application Controller** – centralizuje pobierania i wywoływania komponentów przetwarzania żądań, na przykład poleceń i widoków.
Podstawowe aspekty obsługi żądań, na przykład walidacja, obsługa błędów, autoryzacja i kontrola dostępu mogą zostać w łatwy sposób dołączone do tak stworzonego mechanizmu obsługi żądań klientów.

Mapper – używa obiektu Map do odwzorowania żądań na odpowiednie akcje i widoki. Obiekt Mapper działa jak fabryka.

Map – przechowuje referencje do uchwytów reprezentujących docelowe akcje lub widoki. Mapa może być klasą lub rejestrem.

Target – zasób związany z obsługą konkretnego żądania, który może być poleceniem, widokiem lub arkuszem stylów


Client3
Attributes
Operations
public Client3( )
public ApplicationControler getApplicationControler( )
public void setApplicationControler(ApplicationControler val)
podaje
ApplicationControler
Attributes
Operations
public ApplicationControler( )
public IMapper getIMapper( )
public void setIMapper(IMapper val)
public void wywolanie( )
wywoluje
Target_
Attributes
Operations
public Target_( )
public Target_( )
public void wywolanie_lub_wybor( )
wybiera
dostarcza
FrontController_
Attributes
Operations
public FrontController_( )
InterceptingFilter_
Attributes
Operations
public InterceptingFilter_( )
Mapper
Attributes
Operations
public IMapper( )
public IMap getIMap( )
public void setIMap(IMap val)
public void pobranie_akcji_lub_widoku( )
View_
Attributes
Operations
public View_( )
Command_
Attributes
Operations
public Command_( )
uzywa_
Map
Attributes
Operations
public IMap( )
public void uzycie( )


: Client3
: ApplicationControler
: Mapper
: Map
Target_
public void wywolanie( )
public void pobranie_akcji_lub_widoku( )
public void uzycie( )
public Target_( )
public void wywolanie_lub_wybor( )

Problem 5 – Należy oddzielić widok od logiki związanej z jego przetwarzaniem

Uwagi:

1. Mieszanie logiki sterującej, dostępu do danych i formatowania wewnątrz komponentów widoku prowadzi do problemów z modułowością, ponownym wykorzystaniem komponentów, pielęgnacją i separacją ról programistów. Jest to łamanie zasad projektowania, mówiące o oddzieleniu modelu od widoku i logiki sterującej.
2. Wzorzec **Front Controller** oddziela logikę sterującą, ale nadal istotnym problemem jest oddzielenie modelu i komponentów związanych z widokiem.
3. Szeroko pojęte przetwarzanie żądania wiąże się z dwoma rodzajami zadań: obsługą żądania i przetwarzaniem widoku. Przetwarzanie widoku można podzielić na dwa etapy niezależne, wykorzystujące różną logikę:
   - **przygotowania widoku**: obsługa żądań, zarządzanie akcjami i widokami(przekształcanie żądania na konkretną akcję aplikacji; akcja wywołana powoduje wybór odpowiedniego widoku; następne żądanie jest przekazywane do wybranego widoku),
   - **tworzenia widoku**: widok pobiera odpowiednią zawartość modelu, używając obiektów pomocniczych do uzyskania danych (np. z **Transfer Object**) i ich przekształcenia na odpowiednią postać dla klienta.

Wymagania:

1. Należy korzystać z widoków wykorzystujących szablony, na przykład stron JSP.
2. Należy uniknąć osadzania logiki aplikacji wewnątrz widoków.
3. Należy oddzielić logikę aplikacji od widoku, aby jasno ustalić granice prac programistów i projektantów stron.

Wzorzec **View Helper**

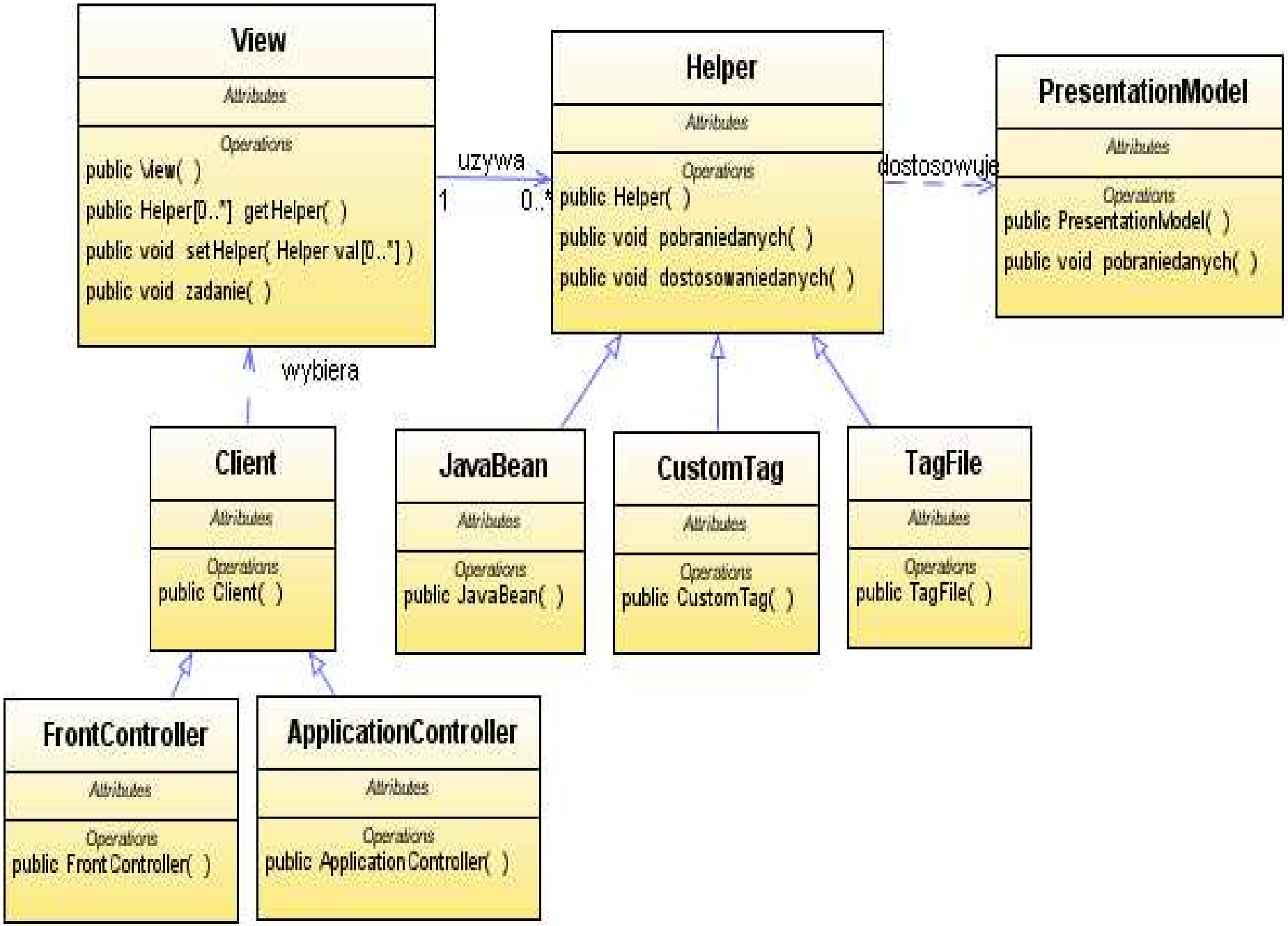
Kod formatujący umieszcza się w widokach, a **logikę przetwarzania** w obiektach pomocniczych.

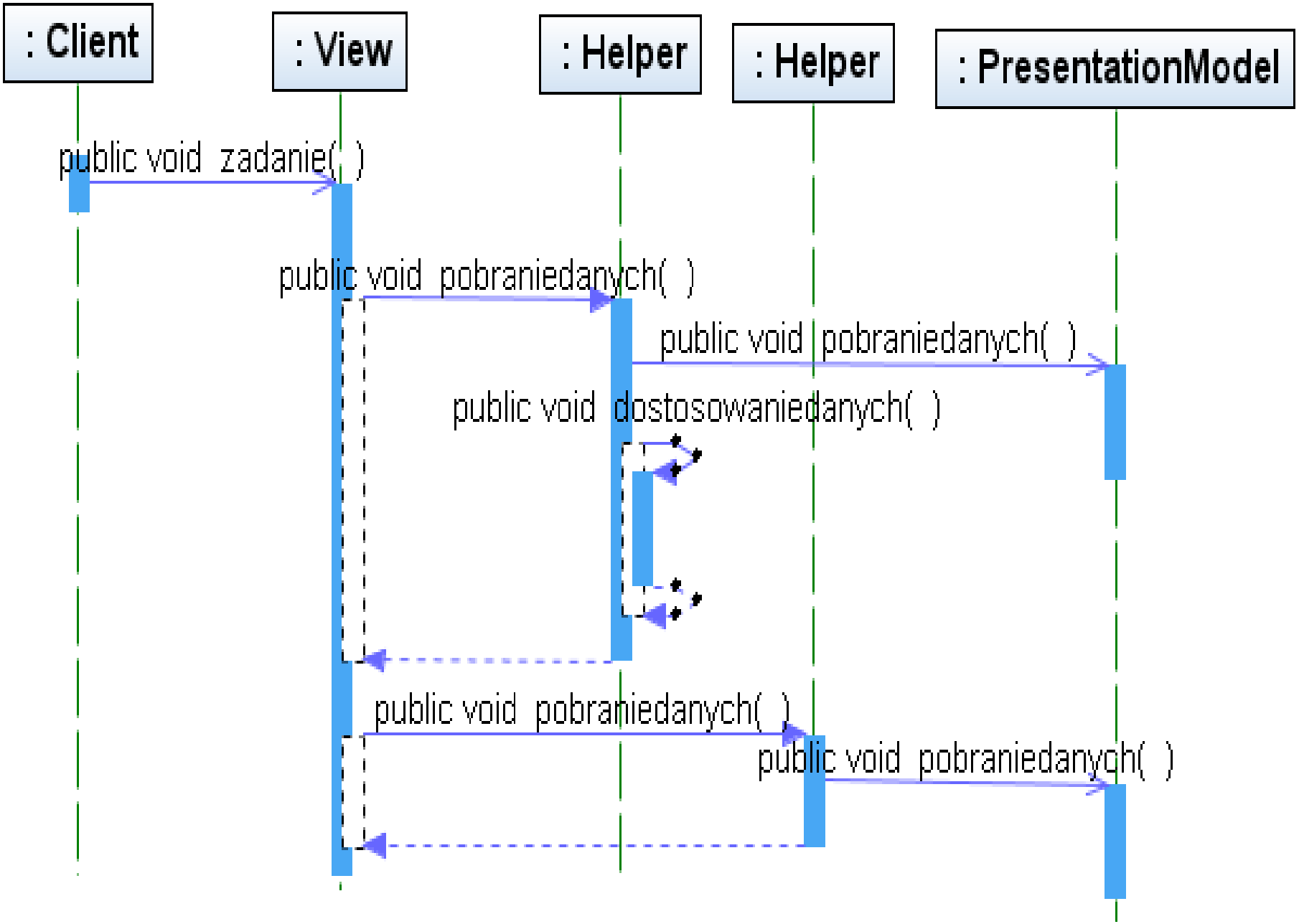
Widok kieruje przetwarzanie do klas pomocniczych, Zaimplementowanych jako zwykłe obiekty Javy (JavaBean), własne znaczniki (CustomTag) lub pliki znaczników (TagFile). Obiekty pomocnicze służą jako pośrednicy między widokiem a modelem, zajmują się przygotowaniem danych do wyświetlenia, na przykład generowania tabel HTML

Inny cel stosowania wzorca:

- Logika biznesowa zostaje zazwyczaj umieszczona w obiekcie modelu, na przykład **TransferObject** lub **BusinessObject**
- Kod dostępu do danych jest umieszczany w obiektach **DAO** zgodnie z wzorcem **Data Access Objects**
- Logika sterująca jest umieszczana we wzorcu **Front Controller** i rozdzielana na odpowiednie polecenia i obiekty pomocnicze
- **PresentationModel** – przechowuje dane otrzymane od usług biznesowych, na podstawie których zostanie wygenerowany widok

- **FrontController** – zajmuje się wstępną obsługą żądania
- **ApplicationController** – wykorzystany do prostego zarządzania widokami, bez zarządzania akcjami aplikacji.
- **View** – zawiera informacje przekazywane klientowi


View
Attributes
Operations
public View( )
public Helper[0..*] getHelper( )
public void setHelper(Helper val[0..*])
public void zadanie( )
uzywa
1
0..*
Helper
Attributes
Operations
public Helper( )
public void pobraniedanych( )
public void dostosowaniedanych( )
dostosowuje
PresentationModel
Attributes
Operations
public PresentationModel( )
public void pobraniedanych( )
wybiera
Client
Attributes
Operations
public Client( )
JavaBean
Attributes
Operations
public JavaBean( )
CustomTag
Attributes
Operations
public CustomTag( )
TagFile
Attributes
Operations
public TagFile( )
FrontController
Attributes
Operations
public FrontController( )
ApplicationController
Attributes
Operations
public ApplicationController( )


: Client
: View
: Helper
: Helper
: PresentationModel
public void zadanie( )
public void pobraniedanych( )
public void pobraniedanych( )
public void dostosowaniedanych( )
public void pobraniedanych( )
public void pobraniedanych( )

Problem 6 – Należy budować widok z modułowych, jednostkowych części składowych, które po połączeniu tworzą złożoną stronę. Poszczególnymi fragmentami zarządzamy niezależnie od siebie.

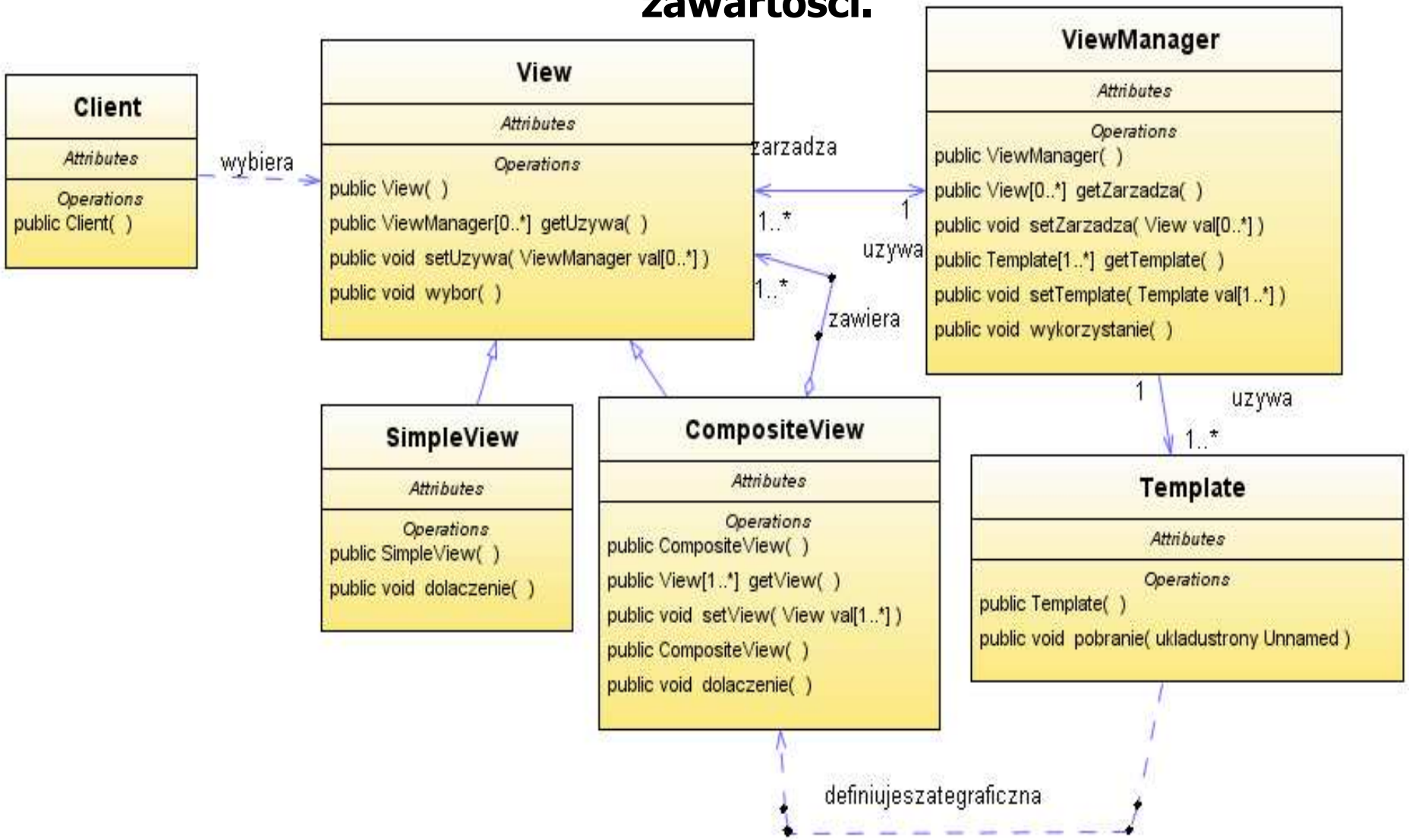
Uwagi:

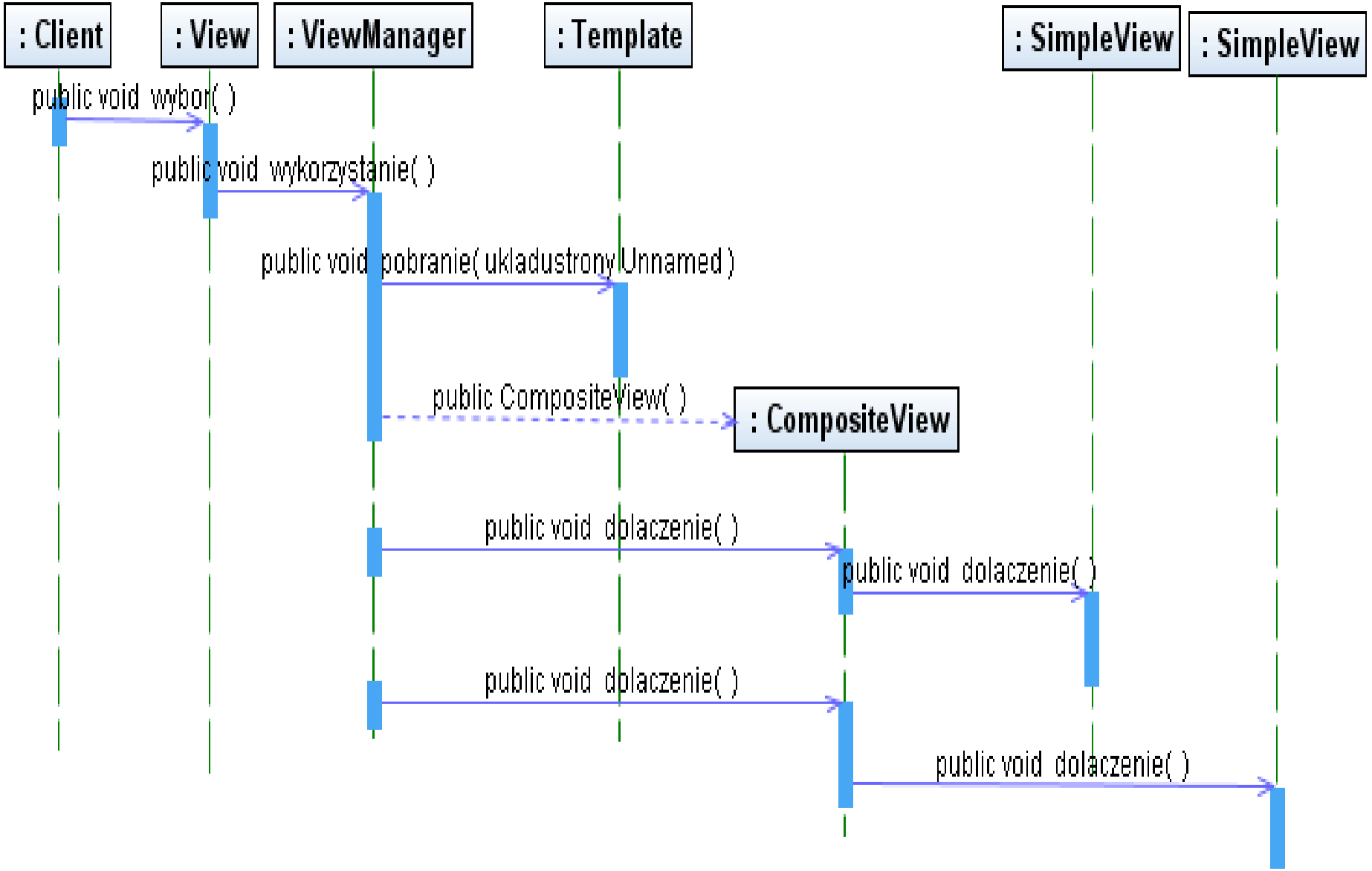
1. Należy unikać powielania kodu w widokach, ponieważ utrudnia to możliwość ponownego wykorzystania i pogarsza modularność oprogramowania

Wymagania:

1. Potrzebne są wspólne podwidoki, na przykład nagłówków, stopek i tabel używanych w wielu widokach. Elementy składowe mogą pojawić się w różnych miejscach dowolnej ze stron.
2. Zawartość znajduje się w podwidokach, które mogą ulegać częstym zmianom lub być udostępniane tylko niektórym użytkownikom, czyli pojawia się problem sterowania dostępem za pomocą ról.
3. Należy uniknąć bezpośrednio powielania i umieszczania podwidoków w wielu widokach, ponieważ utrudnia to znacznie zmiany w układzie stron.

Wzorzec Composite View – składa się z wielu elementarnych podwidoków. Każdy z podwidoków ogólnego szablonu (Template) może zostać dołączany dynamicznie, a układ strony wynikowej jest konfigurowany niezależnie od zawartości.




: Client
: View
: ViewManager
: Template
: SimpleView
: SimpleView
public void wybor( )
public void wykorzystanie( )
public void pobranie(ukladstrony:Unnamed)
public CompositeView( )
: CompositeView
public void dolaczenie( )
public void dolaczenie( )
public void dolaczenie( )
public void dolaczenie( )

Problem 7– Widok zajmuje się obsługą żądania i generuje odpowiedź wykonując niewielką ilość przetwarzania biznesowego

Uwagi:
W pewnych przypadkach przed utworzeniem widoku nie jest konieczne żadne przetwarzanie lub jest go niewiele. Najczęściej są to sytuacje, w których widok jest statyczny lub też jest generowany z istniejącego już modelu prezentacyjnego. Widok w ograniczonym stopniu korzysta z usług biznesowych i dostępu do danych

Wymagania:

1. Widoki są statyczne
2. Widoki są generowane z istniejącego modelu prezentacyjnego
3. Widoki są niezależne od wywołań usług biznesowych
4. Przetwarzania biznesowego jest niewiele

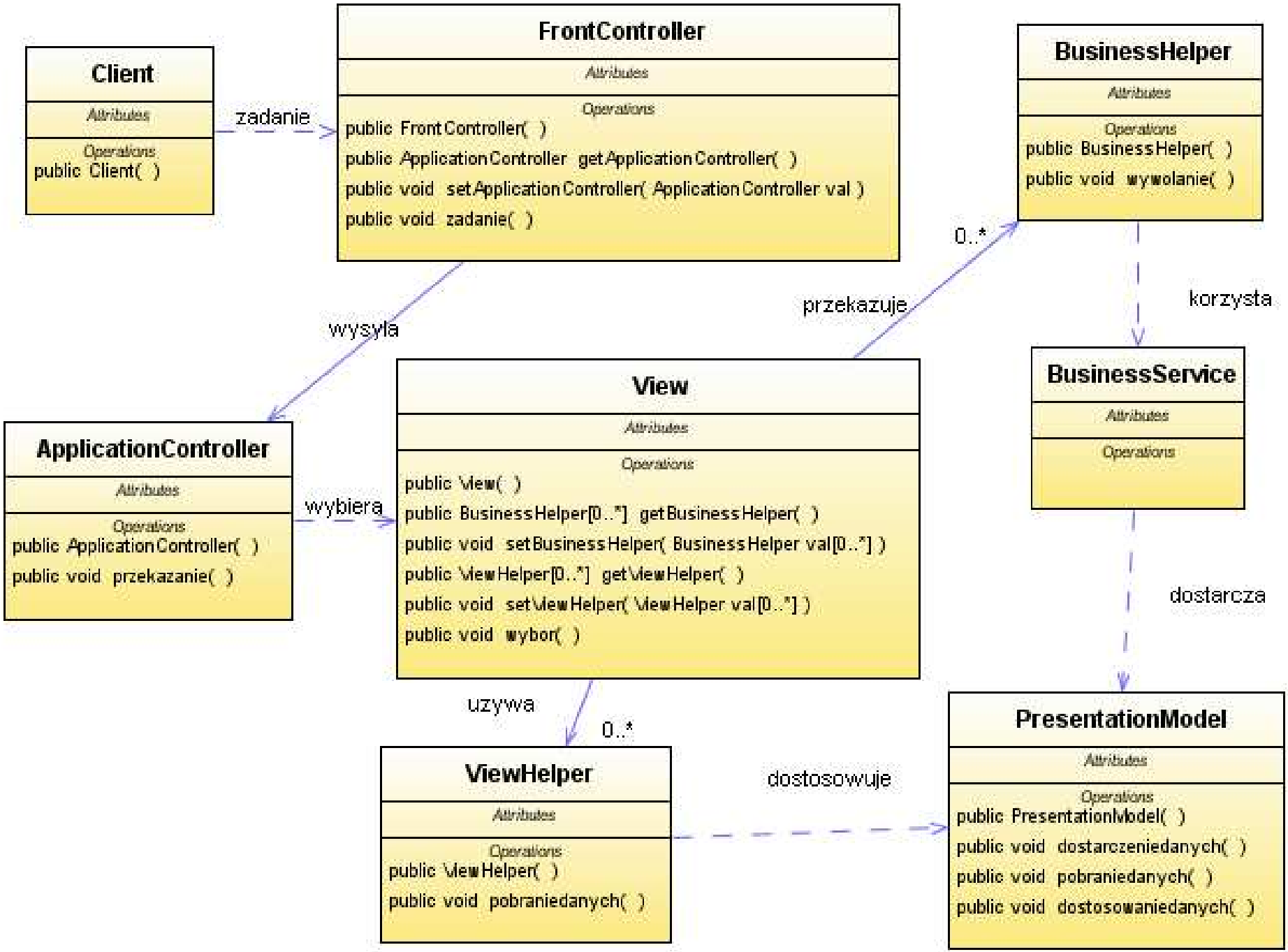
Wzorzec **Dispatcher View** – widoki stanowią początkowy punkt obsługi żądań. Niewielka ilość wymaganego przetwarzania biznesowego jest wykonywana przez widok.

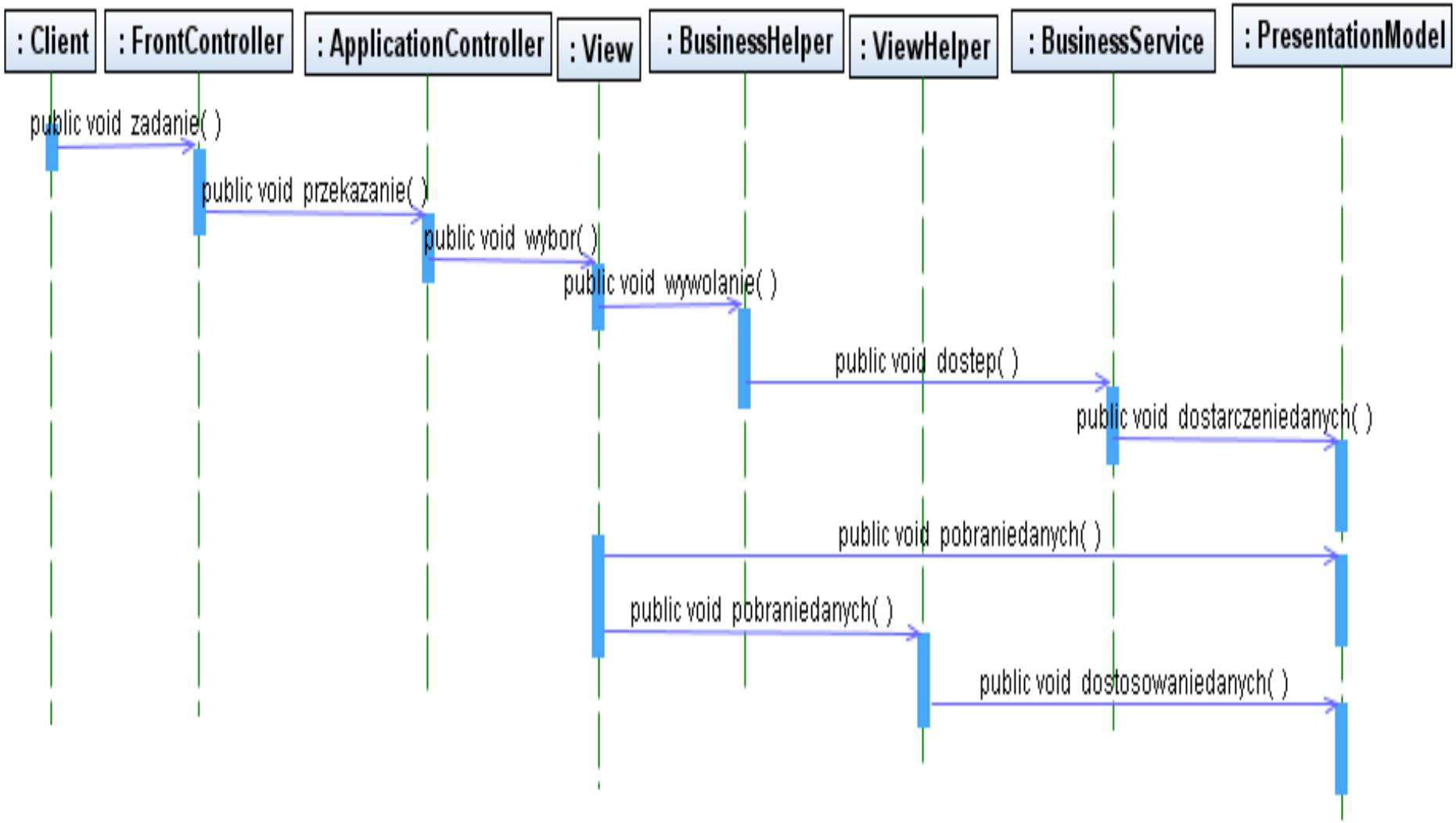
Dwa zastosowania:

1) odpowiedź jest w pełni statyczna – na przykład jest to zwykła strona HTML
2) odpowiedź jest dynamiczna, ale jest w pełni generowana z istniejącego już modelu prezentacyjnego

Elementy wzorca

- **FrontController** – zajmuje się wstępną obsługą żądania
- **ApplicationController** – wykorzystany do prostego zarządzania widokami, bez zarządzania akcjami aplikacji.
- **View** – informacje przekazywane klientowi
- **BusinessHelper, ViewHelper** – pomocnicy wykonujący przetwarzanie żądań związanych z widokami lub kontrelerem.
- **PresentationModel** – przechowuje dane otrzymane od usług biznesowych, na podstawie których zostanie wygenerowany widok
- **BusinessService** – ukrywa szczegóły związane z usługą biznesową. Dostęp do zdalnych usług biznesowych wykonywany jest przez obiekty typu **BusinessDelegate**


Client
Attributes
Operations
public Client( )
zadanie
FrontController
Attributes
Operations
public FrontController( )
public ApplicationController getApplicationController( )
public void setApplicationController(ApplicationController val)
public void zadanie( )
BusinessHelper
Attributes
Operations
public BusinessHelper( )
public void wywolanie( )
0..*
przekazuje
wysyla
korzysta
BusinessService
Attributes
Operations
ApplicationController
Attributes
Operations
public ApplicationController( )
public void przekazanie( )
wybiera
View
Attributes
Operations
public View( )
public BusinessHelper[0..*] getBusinessHelper( )
public void setBusinessHelper(BusinessHelper val[0..*])
public ViewHelper[0..*] getViewHelper( )
public void setViewHelper(ViewHelper val[0..*])
public void wybor( )
dostarcza
uzywa
0..*
ViewHelper
Attributes
Operations
public ViewHelper( )
public void pobraniedanych( )
dostosowuje
PresentationModel
Attributes
Operations
public PresentationModel( )
public void dostarczeniedanych( )
public void pobraniedanych( )
public void dostosowaniedanych( )


: Client
: FrontController
: ApplicationController
: View
: BusinessHelper
: ViewHelper
: BusinessService
: PresentationModel
public void zadanie( )
public void przekazanie( )
public void wybor( )
public void wywolanie( )
public void dostep( )
public void dostarczeniedanych( )
public void pobraniedanych( )
public void pobraniedanych( )
public void dostosowaniedanych( )