Internet Engineering

Tomasz Babczyński, Zofia Kruczkiewicz
Tomasz Kubik

Information systems modelling – UML and service description languages

Choose yourself and new technologies

HUMAN CAPITAL
HUMAN – BEST INVESTMENT!

Wrocław University of Technology

EUROPEAN SOCIAL FUND

Project co-financed from the EU European Social Fund

Design patterns used to build the Presentation Tier

D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

1. The definition of the Presentation Tier – a five-tiered model of logical separation of tasks
2. Basic Presentation Tier design issues
3. Bad practices when designing the Presentation Tier
4. Analysis of basic design issues

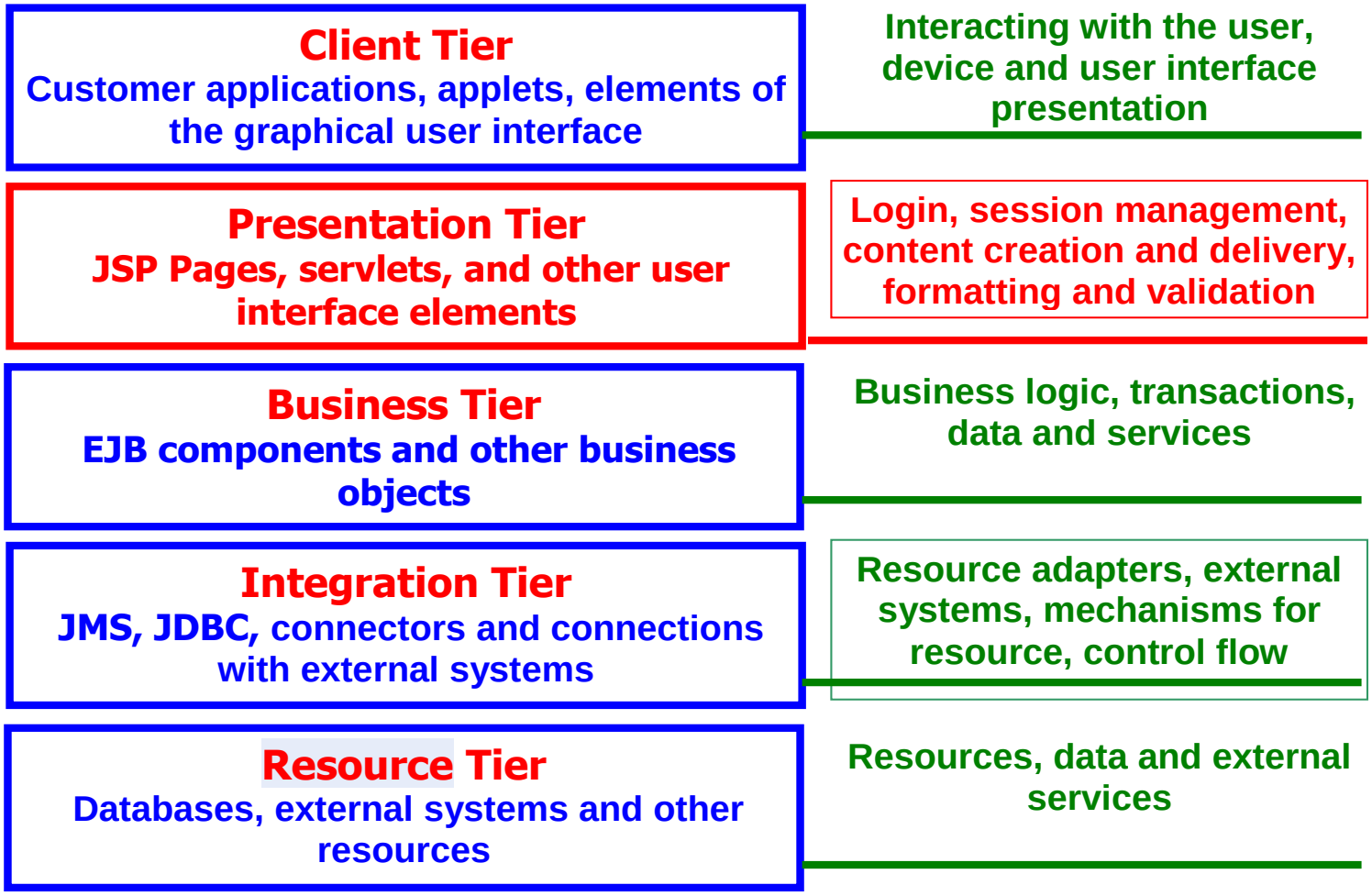
Design patterns used to build the Presentation Tier

D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

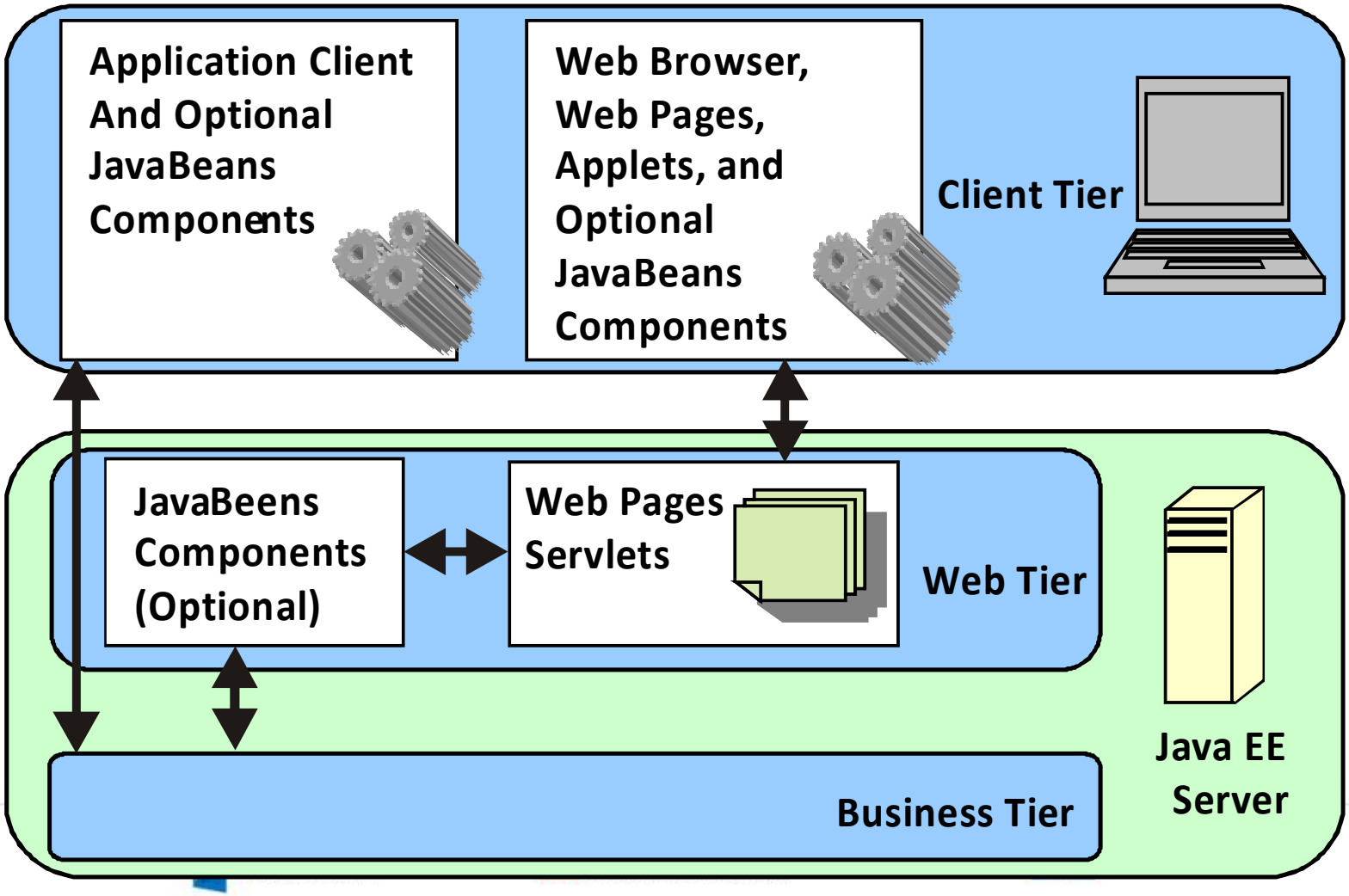
1. The definition of the Presentation Tier – a five-tiered model of logical separation of tasks

Multitiered Information System

by D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns



Web Tier and Java EE Applications (Tutorial Java EE 5)



Design patterns used to build the Presentation Tier

D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

1. The definition of the Presentation Tier – a five-tiered model of logical separation of tasks
2. Basic Presentation Tier design issues

Basic issues of Presentation Tier design

1. Session management
2. Control client access to applications' resources
3. Validation
4. Properties of helper objects
5. Hiding resources from clients, using the configuration of the container

1: Session management

- *session state on the client*: easy to implement, very good performance at a small number of session data (text session data stored in the hidden form fields or cookies)
- *security of the session* - a session state at the client should be encrypted if it should be hidden from the client. This means big trouble for an Enterprise application session due to the large amounts of data.
- *session state in the Presentation Tier* (the difficulty of recovering the client session after server should shutdown) exists until:
  - Session Time has passed
  - The session is cancelled
  - State has been removed from the session.

2: Control client access to applications' resources - authentication and authorization

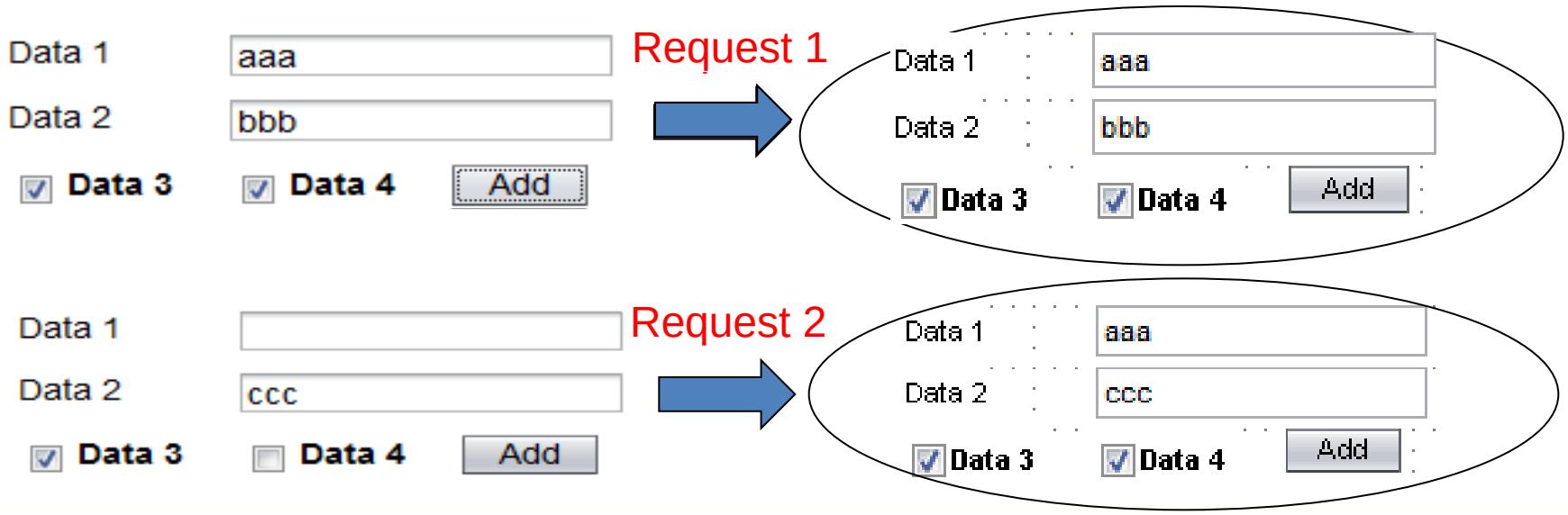
- *view protection*: protection of the entire resources or their portion dependent on the client types, the system state or conditions of errors,
- *protection by the configuration manner*,
- *prevent duplication of forms* in order to avoid repetition of transactions - a synchronizing token stored in the user's session, the direct control of the access to forms.

3: Validation

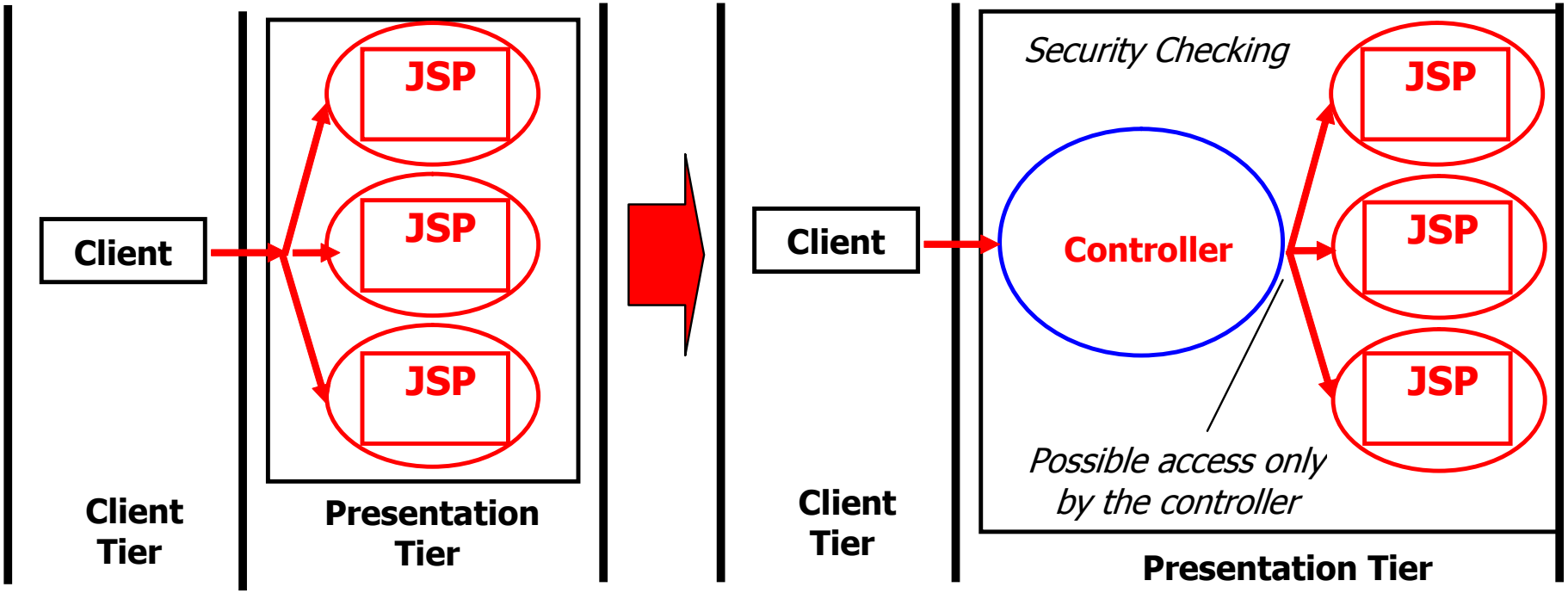
- *validation on the client side* - complements the validation performed on the server should never occur alone
- *validation on the server side*: bound to a form (repetition code) or based on the type of validator classes

4: Properties of helper objects - integrity and consistency

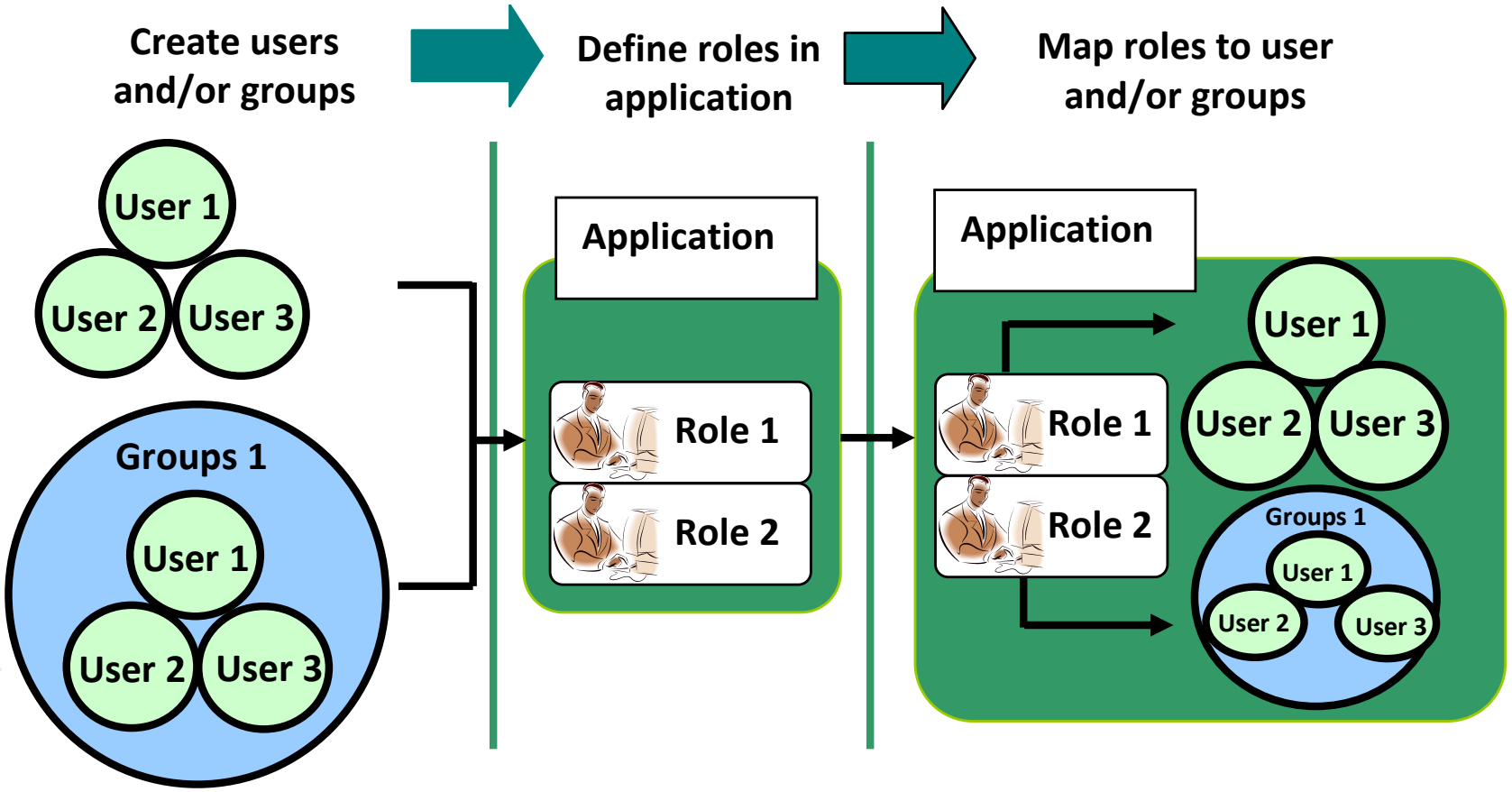
- transfer of the content requests from the Client Tier to the content of these objects



5: Hiding resources from clients, using the configuration of the container (1)



5: Hiding resources from clients, using the configuration of the container (2) - Database Users and Groups, User, Group, Role



- **Declarative security mechanisms**
  - **Descriptors** as an external element of the application specifying the security roles and access requirements, mapped to roles, users, and security policies.

- **Software security mechanisms** - better to express the security model of applications.
  - **API programming mechanisms:**
    - *EJBContext* interface methods
    - *HttpServletRequest* interface methods.
    - Annotations or metadata

  For example,

  @ DeclareRoles ("Customer")

  public class Page1 **extends** AbstractPageBean {//...}

Design patterns used to build the Presentation Tier

D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

1. The definition of the Presentation Tier – a five-tiered model of logical separation of tasks
2. Basic Presentation Tier design issues
3. Bad practices when designing the Presentation Tier

Bad practices (1)

1. Presentation Tier does not carry out the validation, even when validation is performed on the Client Tier
2. Presentation Tier does not carry out data conversion
3. Presentation Tier has available data structures of Business Tiers
4. Presentation Tier data structures are available in other Tiers
5. To transmit data between the Presentation Tier and Business Tier or Integration Tier, are not used auxiliary object class (called an transfer object, which is independent of these tiers)

Bad practices (2)

6. It is not forbidden to send the same form twice and double triggering of the transaction, concerning the same data
7. It does not restrict access to certain forms or parts thereof by authentication (eg. login) and authorization
8. It does not encrypt the sensitive data sent between the client and server
9. There is a scriptlet, inside the dynamically generated HTML code - this code is made available to the application client
10. Mixing control logic, data access and format inside the view components leads to a reduction of modularity, reuse of components, maintenance, and separation of the roles of developers. Design principles are violated ie the separation of model from view and control logic.

Design patterns used to build the Presentation Tier

D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

1. The definition of the Presentation Tier – a five-tiered model of logical separation of tasks
2. Basic Presentation Tier design issues
3. Bad practices when designing the Presentation Tier
4. Analysis of basic design issues

Design cases

1. It should intercept and modify the request and response before and after appropriate processing
2. It should have a centralized access point for handling requests in the Presentation Tier
3. It should avoid using specific protocol information outside its context
4. It should centralize and achieve a modular structure of the management of actions and views of the application
5. The view should be separated from the logic associated with the it processing
6. The view should have the modular structure, built from unit components, which combined make up a complex page. Manage the various parts independently.
7. Major maintenance requests and business logic should be carried out before passing control to the view.
8. The view services the request and generates a response by doing a small amount of business logic

Project co-financed from the EU European Social Fund

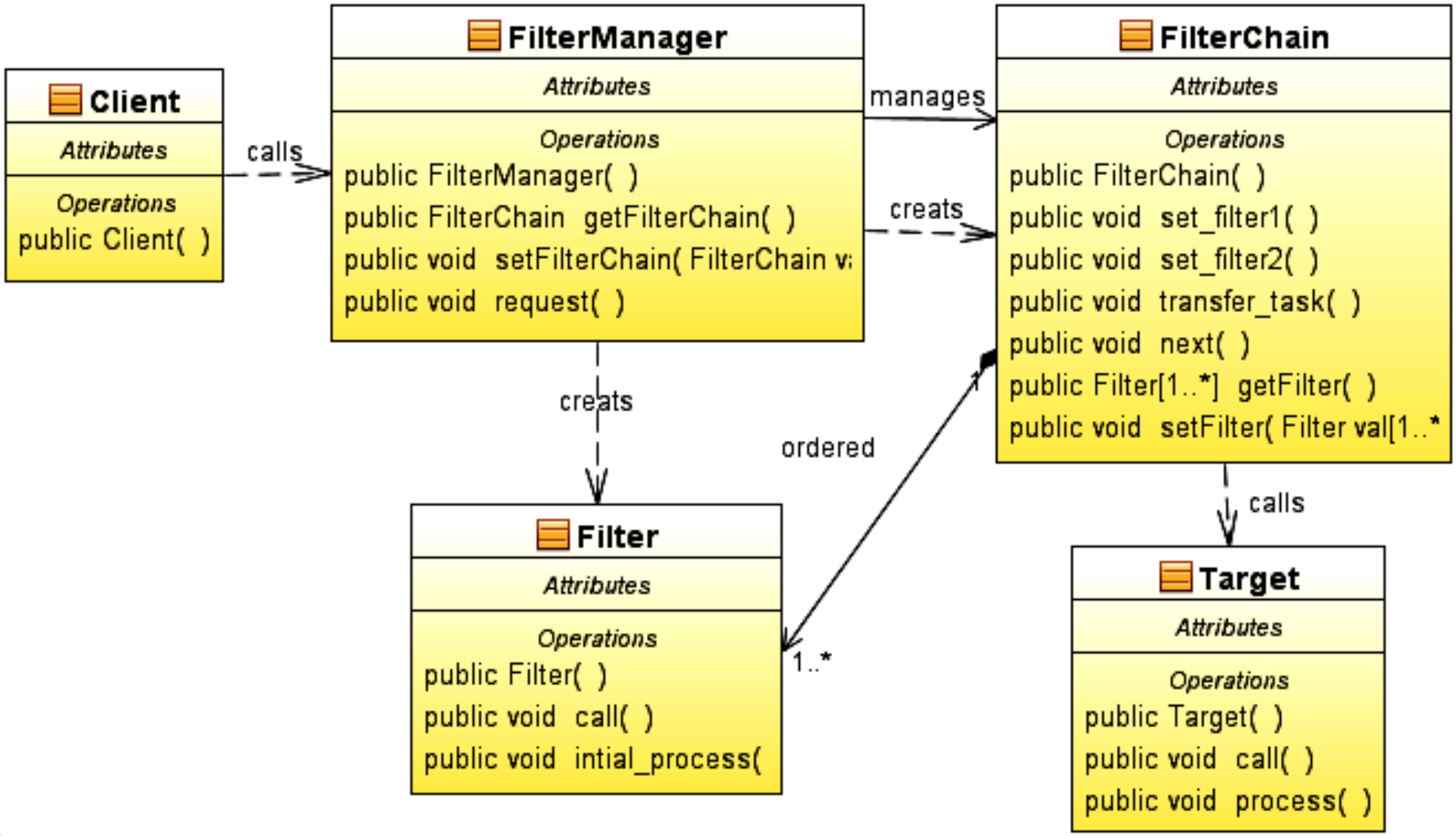
Problem 1 – It should intercept and modify the request and response before and after appropriate processing

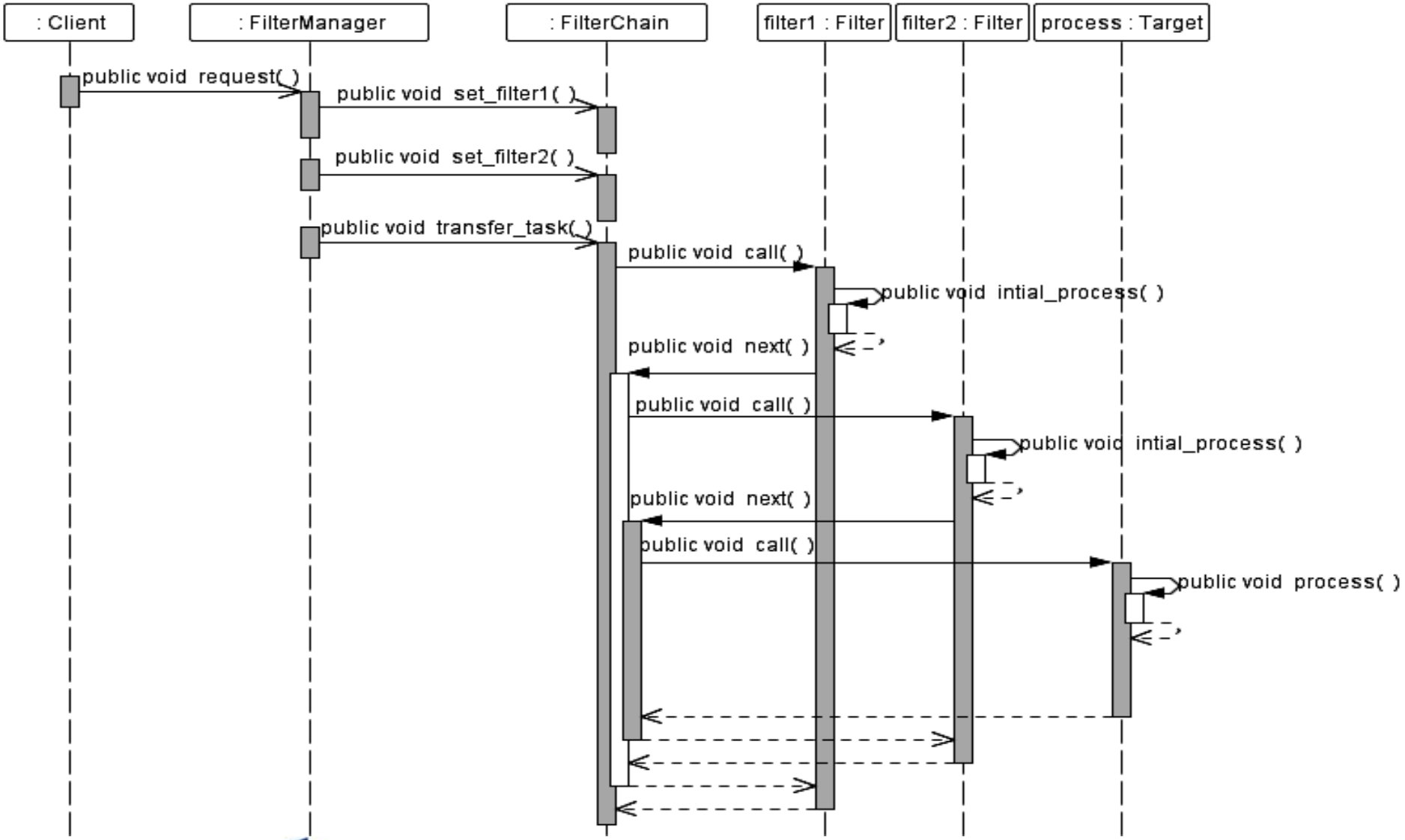
Intercepting Filter

- Is the client linked to the correct session?
- Is the path violated any restrictions?
- Is it supported a particular type of web browser?
- What encryption is used by the client to send data?
- Is the data stream encrypted or compressed?

Requirements

- It should *centralize and process all the requests together*, for example, checking the coding, storing information about each request, the introduction of data compression for the message back sent to the client.
- It *not should involve the code of the pre-processing and final processing of requests with their main code* to make easier to add and remove, for example, new methods of compression.
- Independent components should be used for the processing of initial and final in order to enhance their *capability to re-use*.


Client
Attributes
Operations
public Client( )
calls
FilterManager
Attributes
Operations
public FilterManager( )
public FilterChain getFilterChain( )
public void setFilterChain(FilterChain v
public void request( )
manages
creats
FilterChain
Attributes
Operations
public FilterChain( )
public void set_filter1( )
public void set_filter2( )
public void transfer_task( )
public void next( )
public Filter[1..*] getFilter( )
public void setFilter(Filter val[1..*
creats
ordered
1
1..*
calls
Filter
Attributes
Operations
public Filter( )
public void call( )
public void intial_process(
Target
Attributes
Operations
public Target( )
public void call( )
public void process( )


: Client
: FilterManager
: FilterChain
filter1 : Filter
filter2 : Filter
process : Target
public void request( )
public void set_filter1( )
public void set_filter2( )
public void transfer_task( )
public void call( )
public void intial_process( )
public void next( )
public void call( )
public void intial_process( )
public void next( )
public void call( )
public void process( )

Implementation

- Standard filter
- Custom filter
- Base Filter
- Template filters
- Support for Web services messages
- Custom filter SOAP
- JAX-RPC Filter

Result

- Centralization of control using independent handling procedures
- Improved of reuse
- Declarative and Flexible Configuration
- The small performance of exchange of data between the filters

Problem 2 – It should have a centralized access point for handling requests in the Presentation Tier. **Front Controller**

The lack of centralized access point for services:

- the control code is repeated in many places (eg. views).
- this solution is noither modular nor flexible and makes maintenance difficult

Requirements

- It should avoid *duplication of control logic*
- It should use a *common logic for multiple solutions*
- Processing *logic should be separated from the view*
- It should *centralize and control all access points* to the system

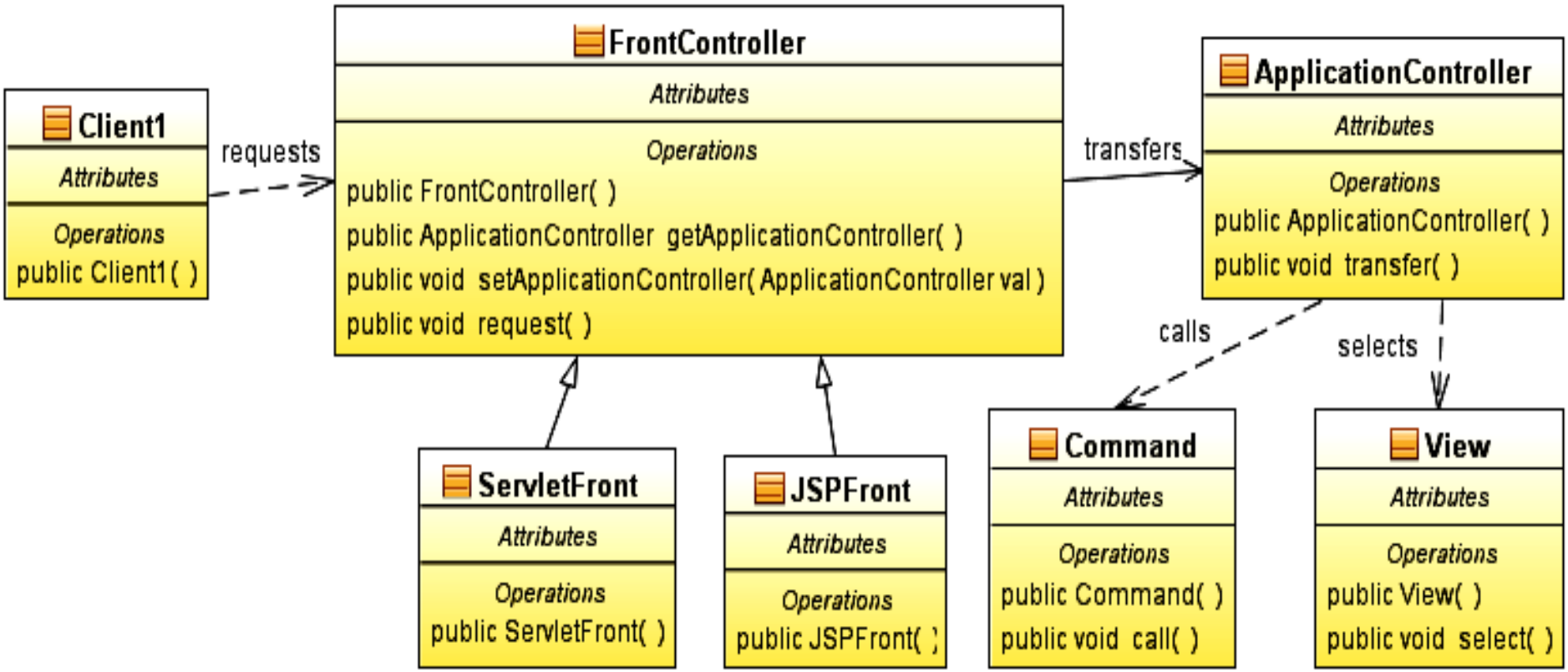
Definitions: *Processing Request*

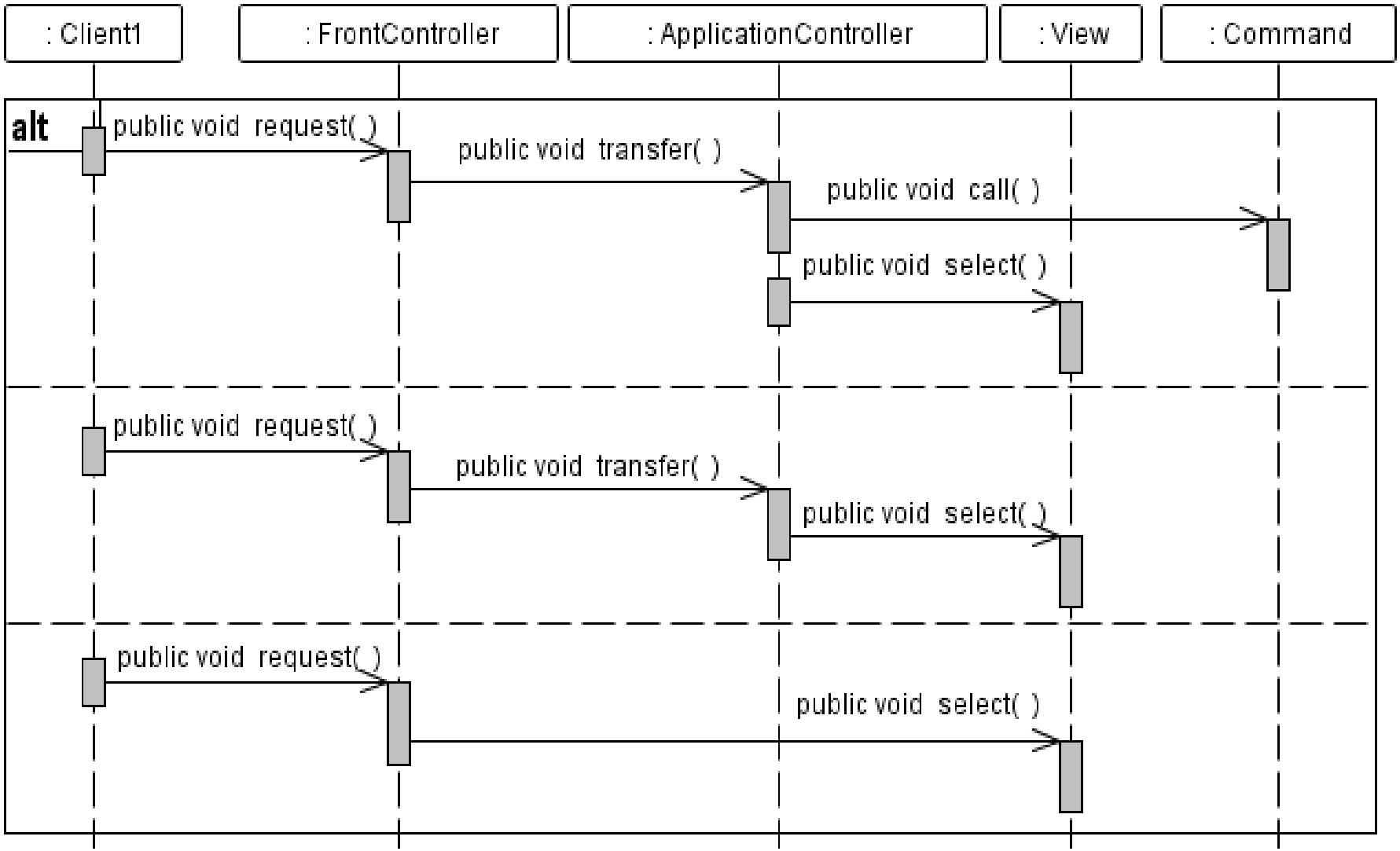
- *Service of the request*
  - protocol support and operations on the context
  - navigation and path selection
  - main processing (actions on the server)
  - transfer of control (dispatch)
- *Processing view*

  - transfer of control to the components of the processing view, after the request service

Result

- Centralization of control
- Improving management of views and the client requests
- Improving opportunities for reuse
- Facilitates the separation of roles among developers


Client1
Attributes
Operations
public Client1( )
requests
FrontController
Attributes
Operations
public FrontController( )
public ApplicationController getApplicationController( )
public void setApplicationController(ApplicationController val)
public void request( )
transfers
ApplicationController
Attributes
Operations
public ApplicationController( )
public void transfer( )
calls
selects
ServletFront
Attributes
Operations
public ServletFront( )
JSPFront
Attributes
Operations
public JSPFront( )
Command
Attributes
Operations
public Command( )
public void call( )
View
Attributes
Operations
public View( )
public void select( )


: Client1
: FrontController
: ApplicationController
: View
: Command
alt
public void request( )
public void transfer( )
public void call( )
public void select( )
public void request( )
public void transfer( )
public void select( )
public void request( )
public void select( )

Implementation

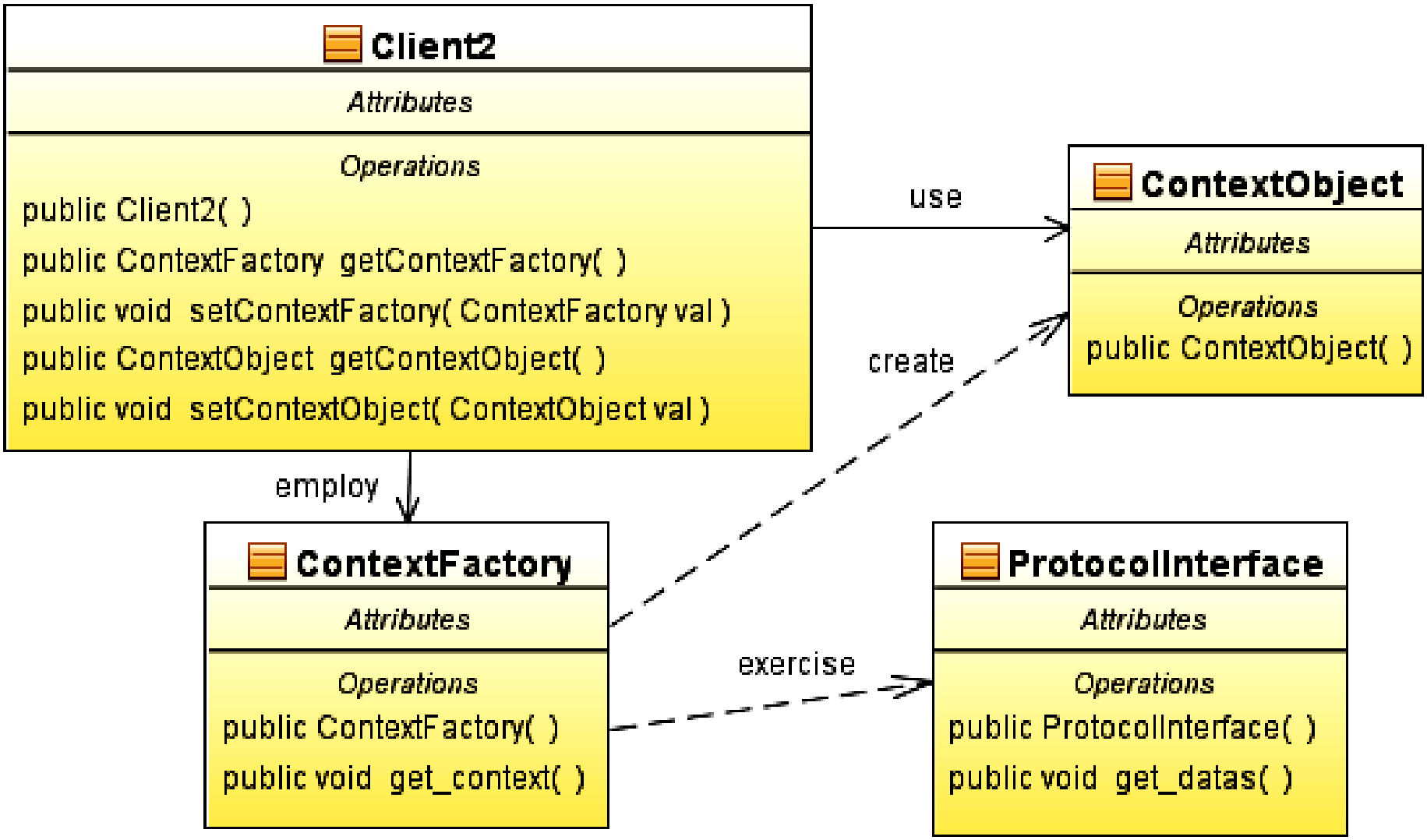
- Servlet receiving requests
- JSP accepting requests
- Command and Controller
- Physical mapping of resources
- Logical mapping of resources
- Repeated mapping of resources
- Selecting a view in the controller
- Base class for accepting the request
- The controller using filters

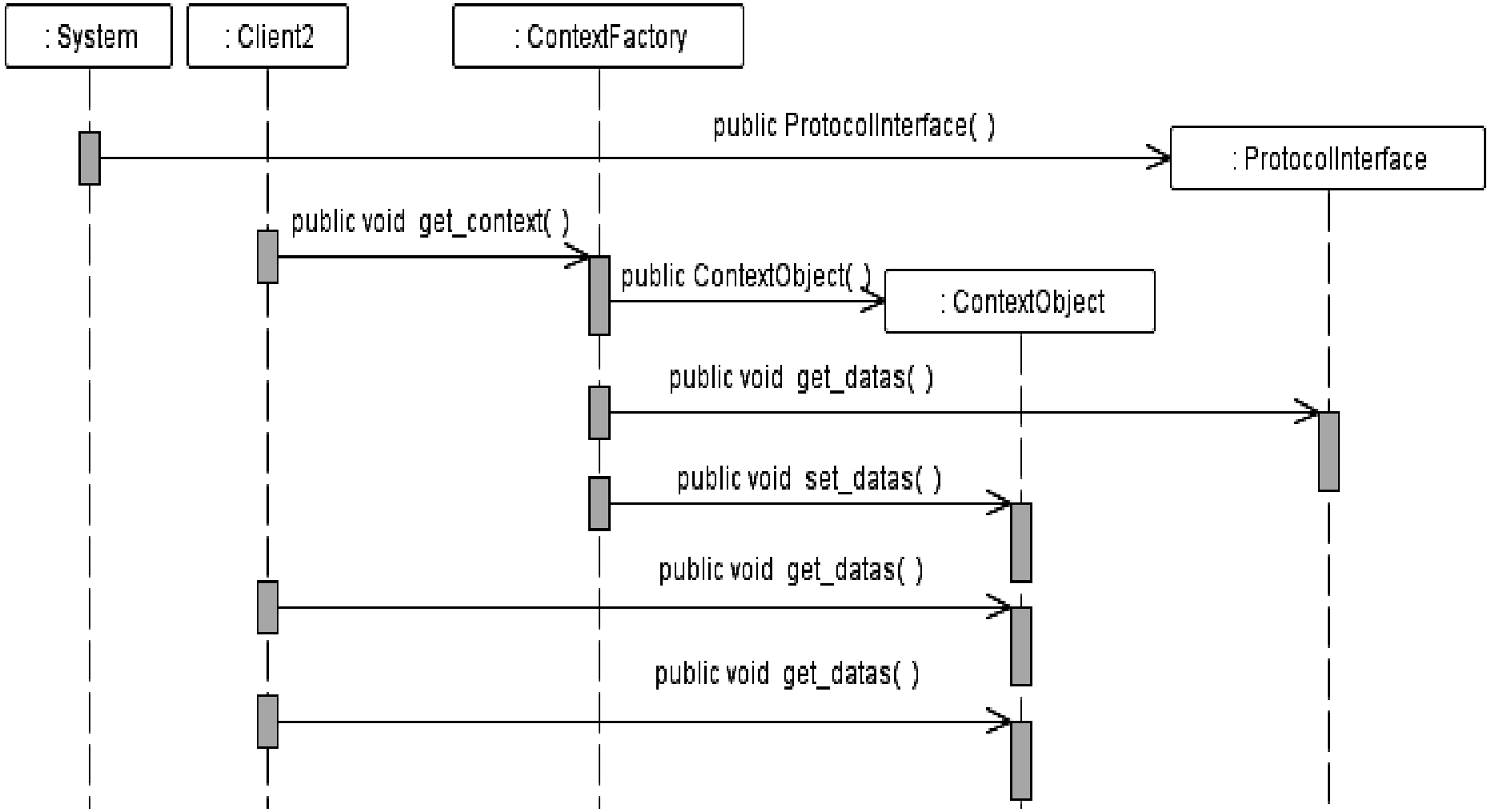
Problem 3 – It should avoid using specific protocol information outside its context. **Context Object**

- The application uses the **appropriate context** with:
  - client requests,
  - the configuration data
  - and data related to safety during the life cycle of request and response objects. These data are taken from.

Requirements

- Components and services require *access to information about systems.*
- *Details of the protocol* should be *separated from the state of application components and services*.
- In the present context, it should *disclose only the necessary elements* of the interface.


Client2
Attributes
Operations
public Client2( )
public ContextFactory getContextFactory( )
public void setContextFactory(ContextFactory val)
public ContextObject getContextObject( )
public void setContextObject(ContextObject val)
use
ContextObject
Attributes
Operations
public ContextObject( )
create
employ
ContextFactory
Attributes
Operations
public ContextFactory( )
public void get_context( )
exercise
ProtocolInterface
Attributes
Operations
public ProtocolInterface( )
public void get_datas( )


: System
: Client2
: ContextFactory
public ProtocolInterface( )
: ProtocolInterface
public void get_context( )
public ContextObject( )
: ContextObject
public void get_datas( )
public void set_datas( )
public void get_datas( )
public void get_datas( )

Implementation

- Context of request
- Context of request in the form of maps
- Context of request in the form of ordinary objects (POJOs)
- Verifying of the context of request
- Configuration context
- Configuration of JSTL
- Security context
- General object of context
  - Factory of contextual objects
  - Autofill contextual objects

Result

- Improvement of maintenance and opportunity of reuse
- Improvement of testability
- Reduction of constraints related to interface changes
- Performance degradation

Problem 4 – It should centralize and achieve a modular structure of the management of actions and views of the application. **Application Controller**

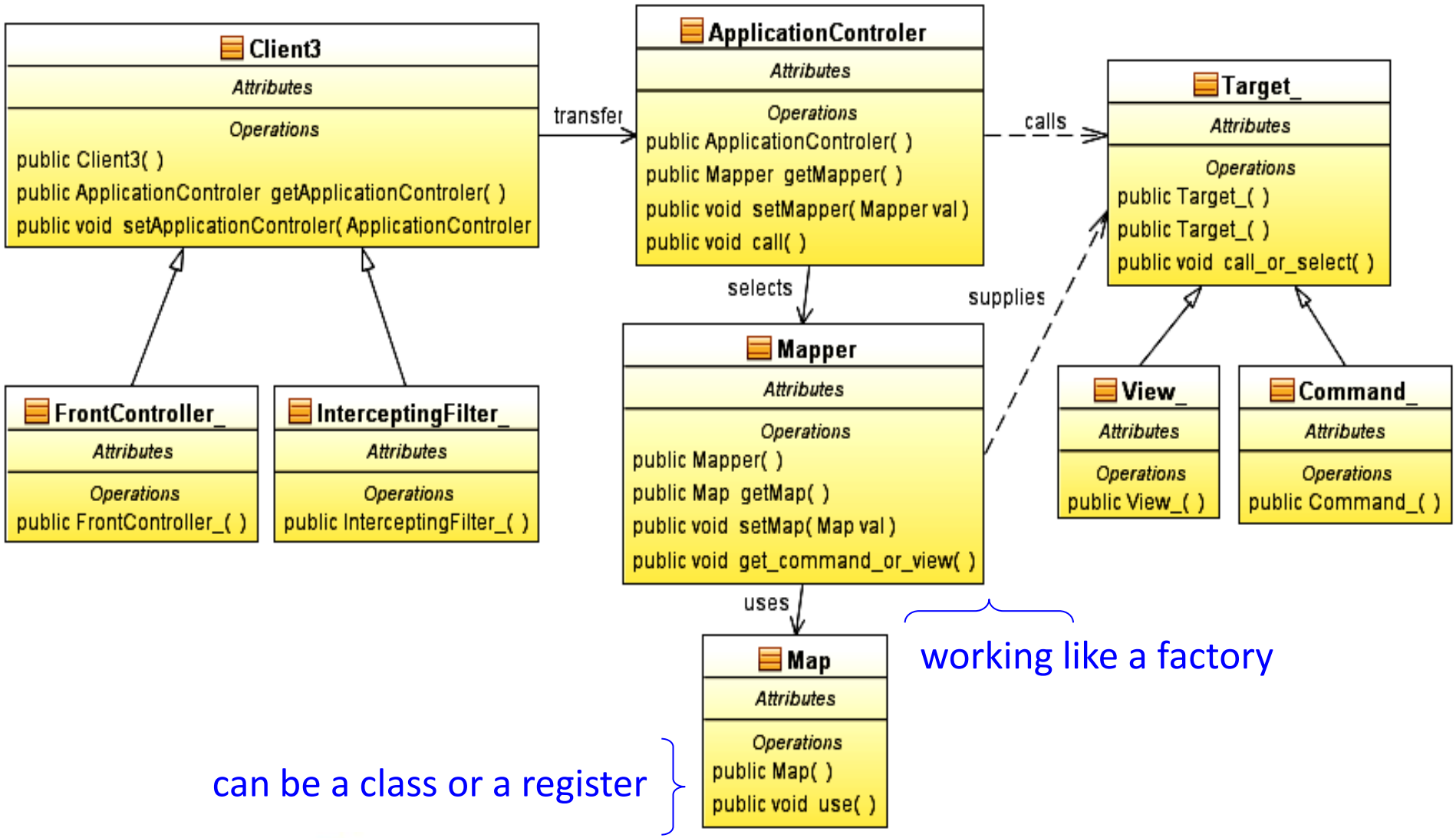
Decisions of Presentation Tier, after receiving the request:

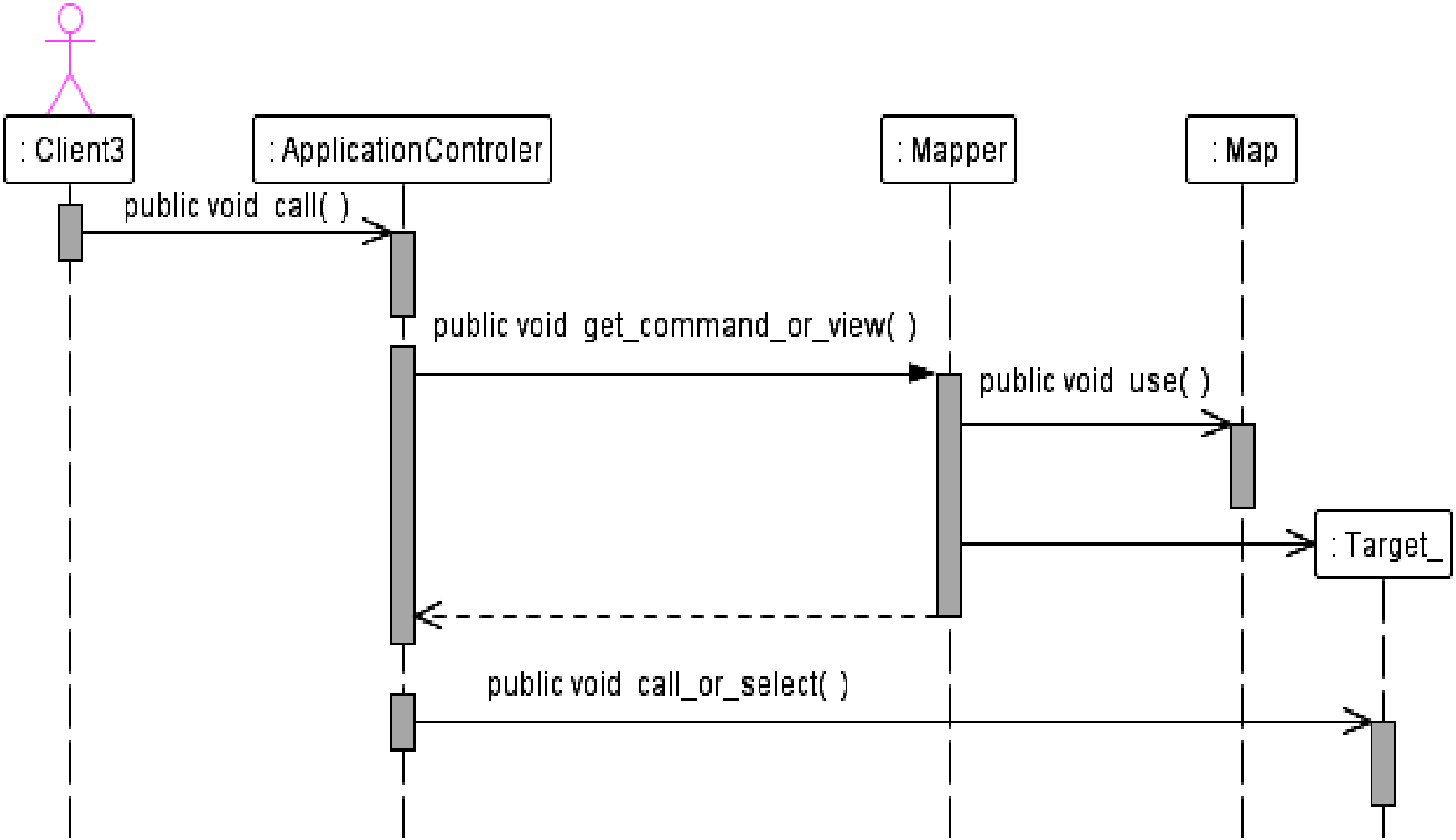
- execution of the appropriate actions of the required service
- it should localize and use an appropriate view
- basic aspects of service requests, for example, validation, error handling, authentication and access control can be easily attached to the client requests

Requirements

- *A repeated code fragment for management of actions and views*
- *It improves expansion of request services*, such as the ability to incrementally add more implementations of use cases.
- *It increases the modularity of code* to facilitate the expansion of application functionality and ease to test code and individual requests

Project co-financed from the EU European Social Fund


Client3
Attributes
Operations
public Client3( )
public ApplicationControler getApplicationControler( )
public void setApplicationControler(ApplicationControler
transfer
ApplicationControler
Attributes
Operations
public ApplicationControler( )
public Mapper getMapper( )
public void setMapper(Mapper val)
public void call( )
calls
Target_
Attributes
Operations
public Target_( )
public Target_( )
public void call_or_select( )
selects
supplies
Mapper
Attributes
Operations
public Mapper( )
public Map getMap( )
public void setMap(Map val)
public void get_command_or_view( )
FrontController_
Attributes
Operations
public FrontController_( )
InterceptingFilter_
Attributes
Operations
public InterceptingFilter_( )
View_
Attributes
Operations
public View_( )
Command_
Attributes
Operations
public Command_( )
uses
Map
Attributes
Operations
public Map( )
public void use( )
working like a factory
can be a class or a register


: Client3
: ApplicationControler
: Mapper
: Map
: Target_
public void call( )
public void get_command_or_view( )
public void use( )
public void call_or_select( )

Implementation

- Service of the command
- Service of the view
- Transformation service
- Control of flow and navigation
- Web service
  - self service of SOAP messages
  - service of JAX-RPC messages or web service

Result

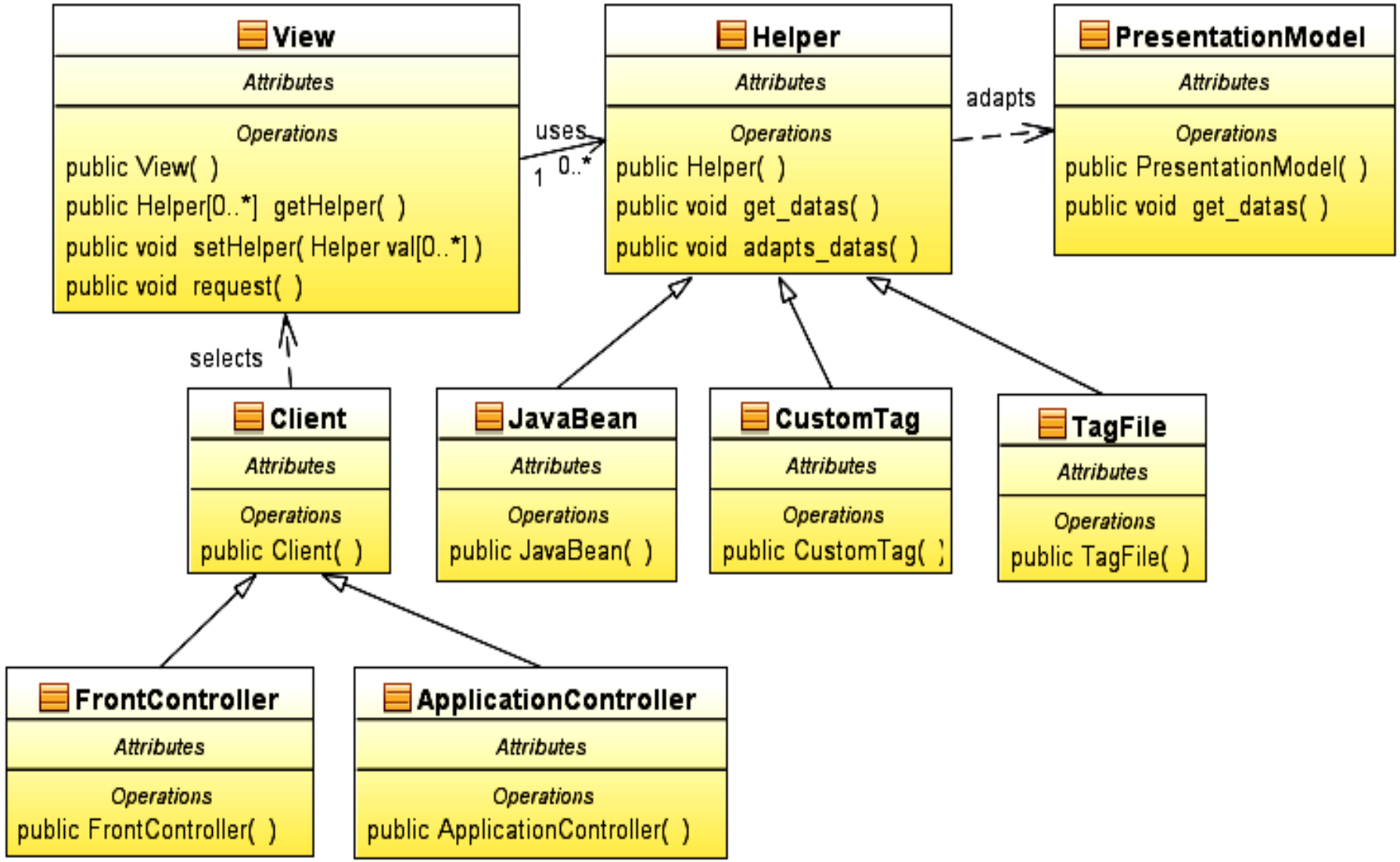
- Improved:
  - modularity
  - reuse
  - expansion

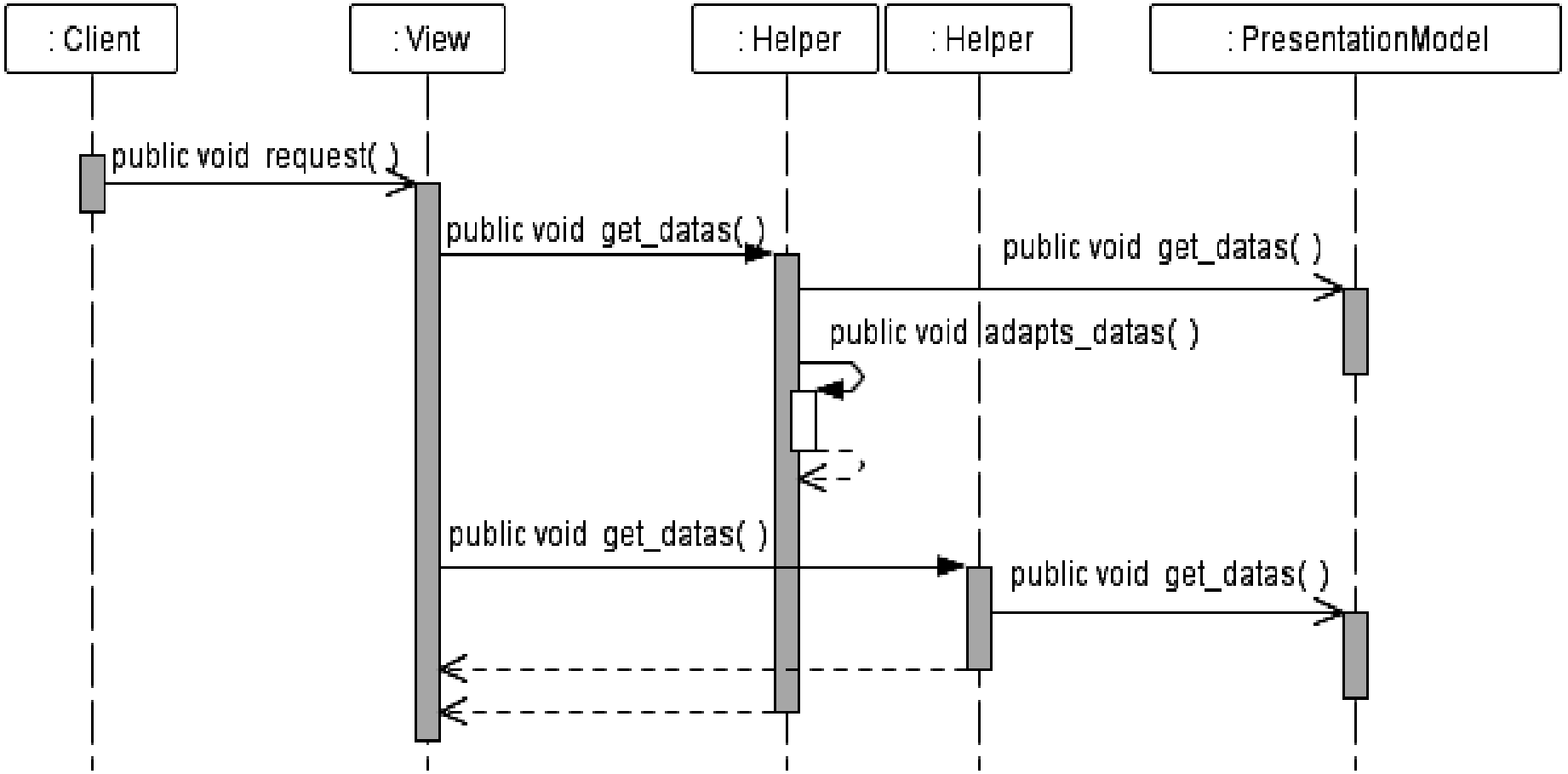
Problem 5 – The view should be separated from the logic associated with the its processing. **View Helper**

1. *The Front Controller* pattern separates the logic control from the view.
2. Processing of view can be divided into two independent stages, using different logics:
   - preparation of view: service of requests, management of actions and views
   - creating a view: a view retrieves the appropriate content of model, using the auxiliary objects to obtain data

Requirements

- It should use *templates of views*, such as JSPs.
- It should *avoid embedding application logic inside the view* (avoiding the use of scriptlets).
- Application logic should be separated from the view, to clearly *separate programmers and web designers resposibilities*.


View
Attributes
Operations
public View( )
public Helper[0..*] getHelper( )
public void setHelper(Helper val[0..*])
public void request( )
uses
1
0..*
Helper
Attributes
Operations
public Helper( )
public void get_datas( )
public void adapts_datas( )
adapts
PresentationModel
Attributes
Operations
public PresentationModel( )
public void get_datas( )
selects
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 request( )
public void get_datas( )
public void get_datas( )
public void adapts_datas( )
public void get_datas( )
public void get_datas( )

Implementation

- Views using templates
- View controller
- Auxiliary *JavaBean* object
- Custom tags
- Assistant in the form of a file Tag
- Auxiliary *BusinessDelegate* object

Result

- *Improved modularity, reuse and facilitating software maintenance* (avoiding Java scriptlets, and HTML code, which is contained in code of helper component)
- Improved *separation of roles*
- *Facilitate the testing*
- The use of auxiliary objects are not always different from the use of Java scriptlets

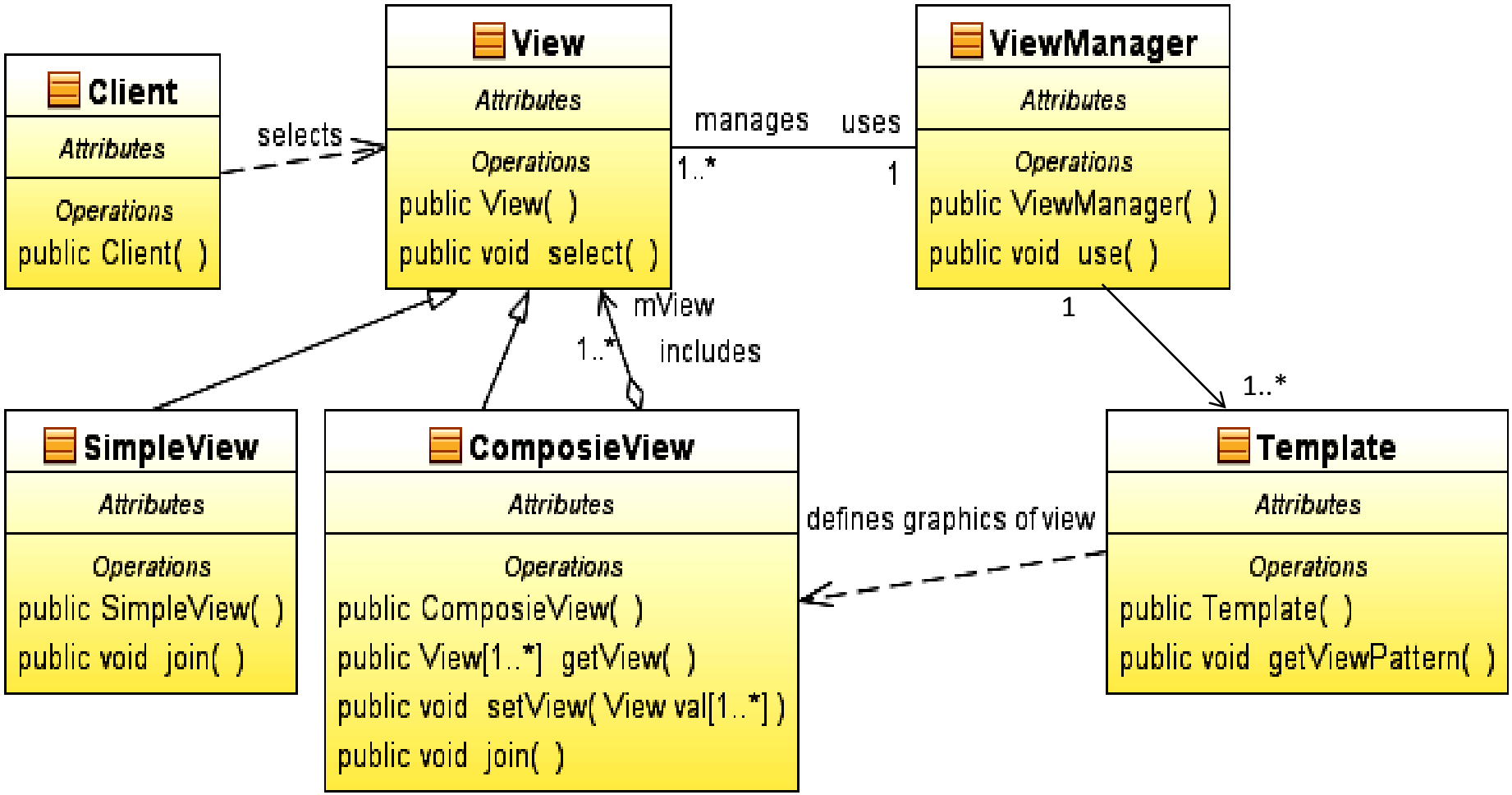
Problem 6 – The view should have modular a structure, built from unit components, which combined make up together a complex page. Manage the various parts independently. **Composite View**

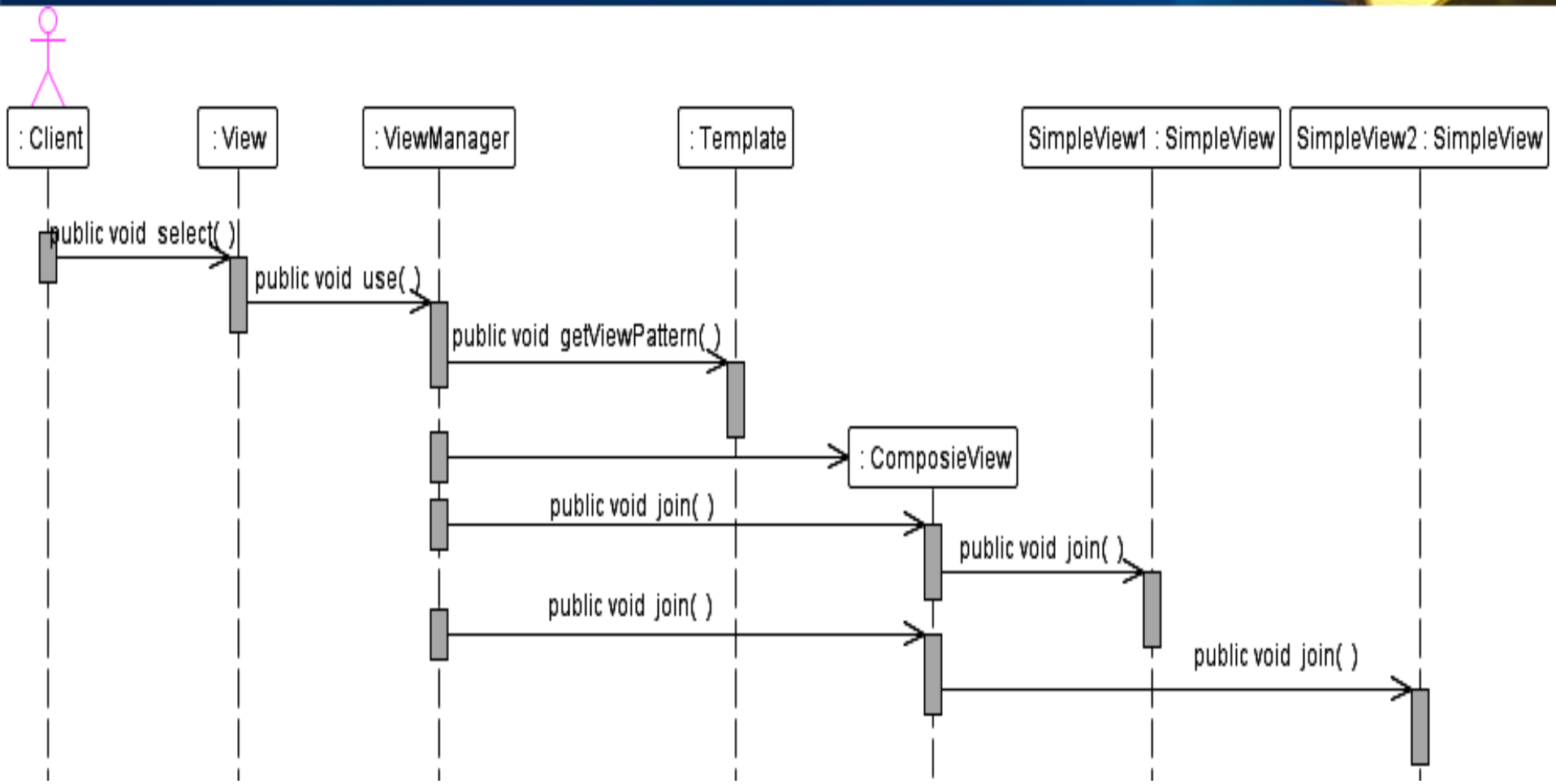
Avoid duplication of code in your views, because:

- it hinders the possibility of code reuse, which reduces the modularity of software - software quality is deteriorating, represented by software reuse
- duplicate code is harder to manage – it is deteriorating the quality of the software, represented by the maintainability of software.

Requirements

- *Subviews should be common (as the Template)*, such as headers, footers, and tables used in many views. Components can appear in different places and pages.
- The *content of subviews are subject to frequent change*, or be made available only to some users,
- It should *avoid duplication and direct placement* of subviews in many views,


Client
Attributes
Operations
public Client( )
selects
View
Attributes
Operations
public View( )
public void select( )
manages
uses
1..*
1
ViewManager
Attributes
Operations
public ViewManager( )
public void use( )
1
1..*
mView
1..*
includes
SimpleView
Attributes
Operations
public SimpleView( )
public void join( )
ComposieView
Attributes
Operations
public ComposieView( )
public View[1..*] getView( )
public void setView(View val[1..*])
public void join( )
defines graphics of view
Template
Attributes
Operations
public Template( )
public void getViewPattern( )


: Client
: View
: ViewManager
: Template
SimpleView1 : SimpleView
SimpleView2 : SimpleView
public void select( )
public void use( )
public void getViewPattern( )
: ComposieView
public void join( )
public void join( )
public void join( )
public void join( )

Implementation

- Managing views with:
  - JavaBean components
  - standard markers
  - custom tags
  - transformation
- Early binding of resources
- Late binding of resources

Result

- Improvement of modularity and opportunity of reuse
- Adding a control scheme based on roles or rules
- Facilitating maintenance of the code
- Handicap code management (danger of mismatches of subview)
- Performance degradation when generating the subview

Problem 7: Should be carried out major maintenance requests and business logics before passing control to the view. **Service to Worker**

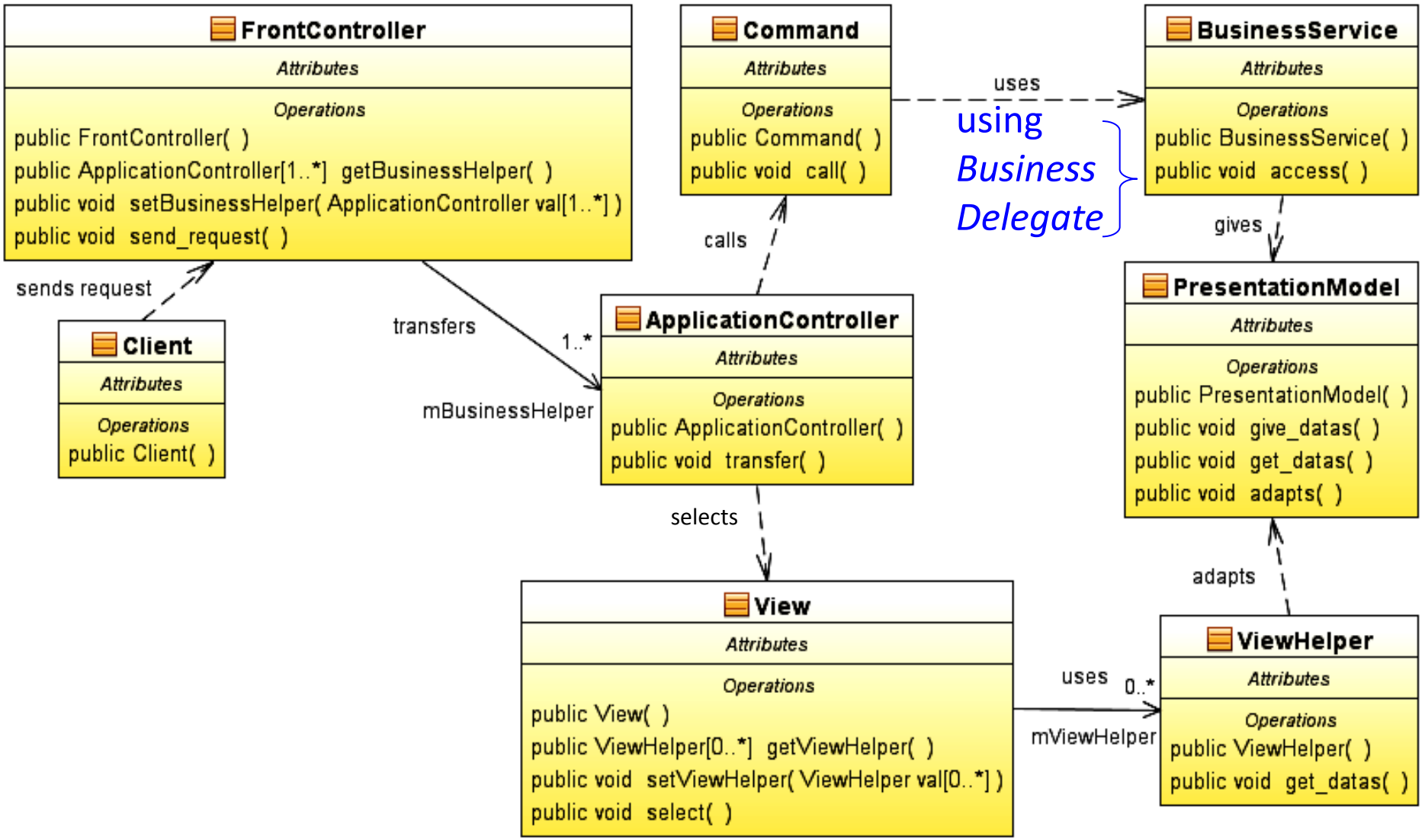
It should be taken into account:

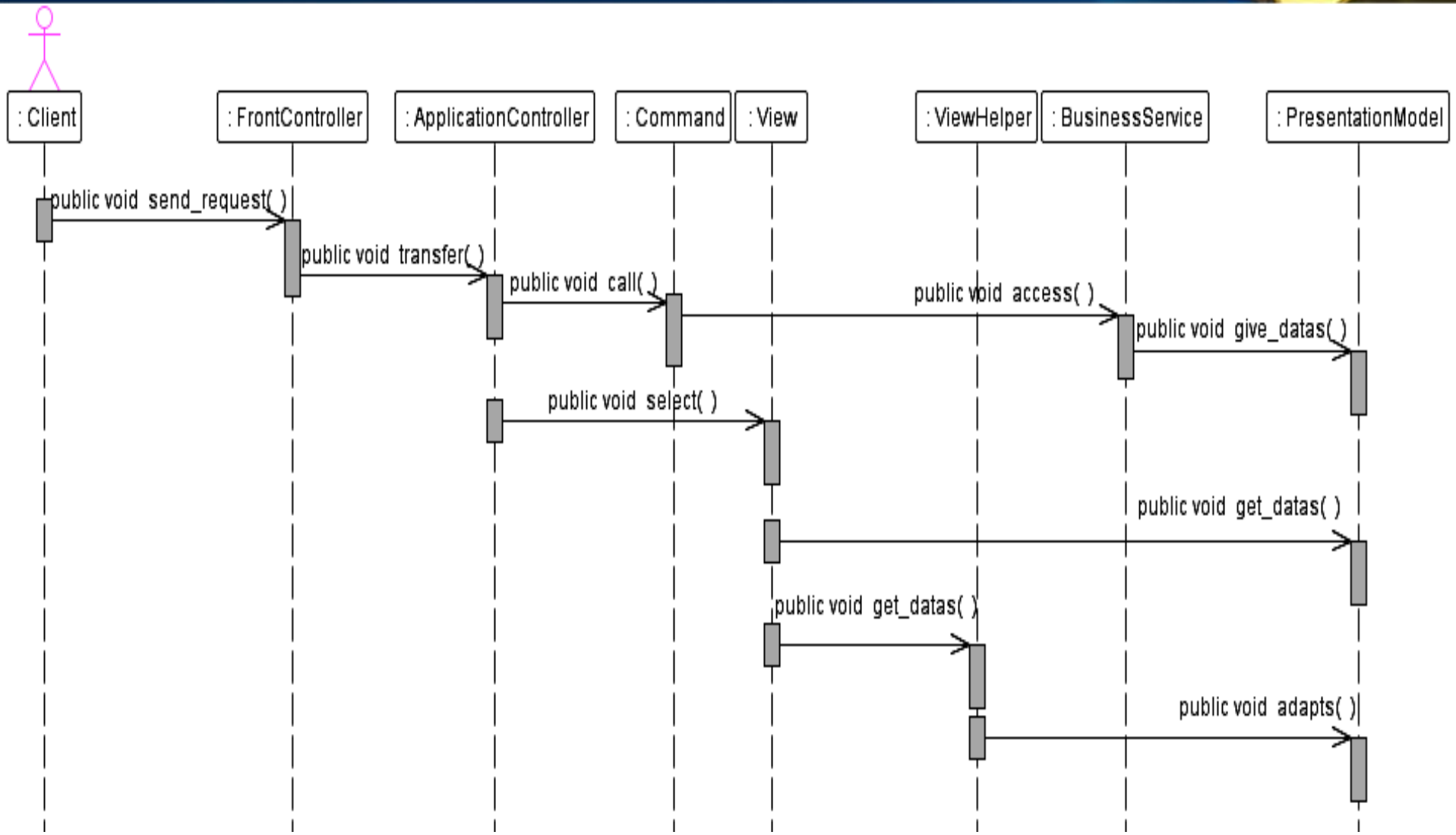
- Complexity of control logic
- Dynamics of answers
- Complexity of business logic and models

Requirements

- Perform business logic to handle the request and retrieve data that will be used to *generate a dynamic response*
- The contents of the *view depends on the response* received after performing the business services
- It should be used the library or *skeleton of presentation*.
- The need of the *most common method of designing the Presentation Tier*

Project co-financed from the EU European Social Fund


FrontController
Attributes
Operations
public FrontController( )
public ApplicationController[1..*] getBusinessHelper( )
public void setBusinessHelper(ApplicationController val[1..*])
public void send_request( )
Command
Attributes
Operations
public Command( )
public void call( )
BusinessService
Attributes
Operations
public BusinessService( )
public void access( )
uses
using
Business
Delegate
gives
calls
sends request
Client
Attributes
Operations
public Client( )
transfers
1..*
mBusinessHelper
ApplicationController
Attributes
Operations
public ApplicationController( )
public void transfer( )
PresentationModel
Attributes
Operations
public PresentationModel( )
public void give_datas( )
public void get_datas( )
public void adapts( )
selects
adapts
View
Attributes
Operations
public View( )
public ViewHelper[0..*] getViewHelper( )
public void setViewHelper(ViewHelper val[0..*])
public void select( )
uses
0..*
mViewHelper
ViewHelper
Attributes
Operations
public ViewHelper( )
public void get_datas( )


: Client
: FrontController
: ApplicationController
: Command
: View
: ViewHelper
: BusinessService
: PresentationModel
public void send_request( )
public void transfer( )
public void call( )
public void access( )
public void give_datas( )
public void select( )
public void get_datas( )
public void get_datas( )
public void adapts( )

Implementation

- Servlet receiving requests
- JSP page receiving request
- Views using templates
- View-based controller
- Auxiliary JavaBean components
- Auxiliary own tags
- Selecting a view in the controller

Result

- Centralizes control and improves modularity, reusability, and maintenance eases
- A proper separation between the developers roles

Design patterns used to build the presentation Tier

D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

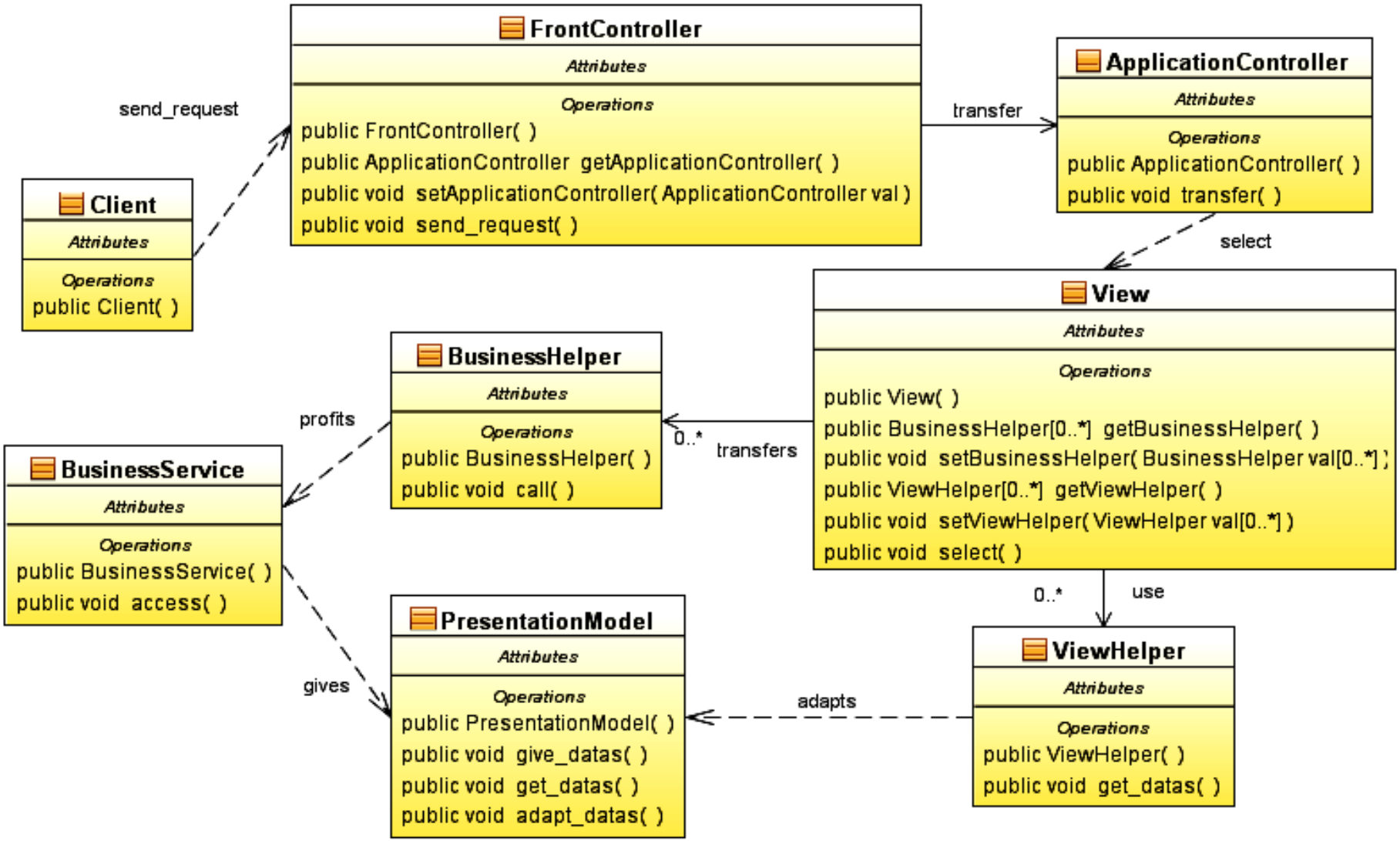
1. The definition of the Presentation Tier – a five-tiered model of logical separation of tasks
2. Basic Presentation Tier design issues
3. Bad practices when designing the Presentation Tier
4. Analysis of basic design issues
5. Appendix

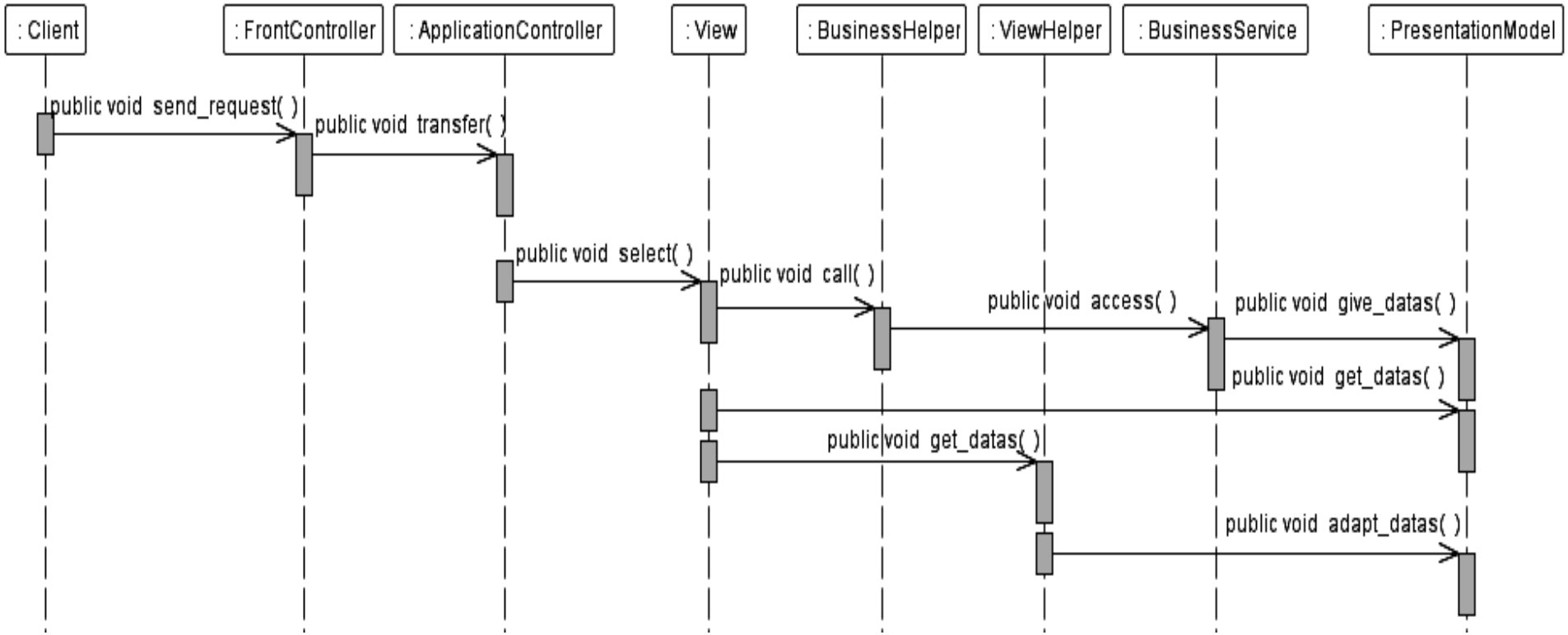
Problem 8 – A view can service request and generate responses by doing a small amount of business logics.

Dispatcher View

Requirements

- The need of the second most common method of designing the Presentation Tier, where the *View* component plays the main role
- Views are static (a plain HTML page)
- Dynamic views are generated from an existing presentation model
- Views are the initial points of requests services.
- Views are independent of the calls of business services
- A few business processing


FrontController
Attributes
Operations
public FrontController( )
public ApplicationController getApplicationController( )
public void setApplicationController(ApplicationController val)
public void send_request( )
send_request
transfer
ApplicationController
Attributes
Operations
public ApplicationController( )
public void transfer( )
select
Client
Attributes
Operations
public Client( )
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 select( )
BusinessHelper
Attributes
Operations
public BusinessHelper( )
public void call( )
profits
0..*
transfers
BusinessService
Attributes
Operations
public BusinessService( )
public void access( )
0..*
use
PresentationModel
Attributes
Operations
public PresentationModel( )
public void give_datas( )
public void get_datas( )
public void adapt_datas( )
gives
adapts
ViewHelper
Attributes
Operations
public ViewHelper( )
public void get_datas( )


: Client
: FrontController
: ApplicationController
: View
: BusinessHelper
: ViewHelper
: BusinessService
: PresentationModel
public void send_request( )
public void transfer( )
public void select( )
public void call( )
public void access( )
public void give_datas( )
public void get_datas( )
public void get_datas( )
public void adapt_datas( )

Implementation

- Servlet receiving requests
- JSP page receiving request
- Views using templates
- View-based controller
- Auxiliary JavaBean components
- Auxiliary own tags
- Selecting a view in the controller

Result

- It is used in the *presentation skeletons* and the library (e.g. JSTL)
- It may result in *poor separation* of a view from a model and control logic
- Allows *the separation of processing logic from the view*, and facilitates code reuse

JavaServer Faces Standard Request-Response Life Cycle