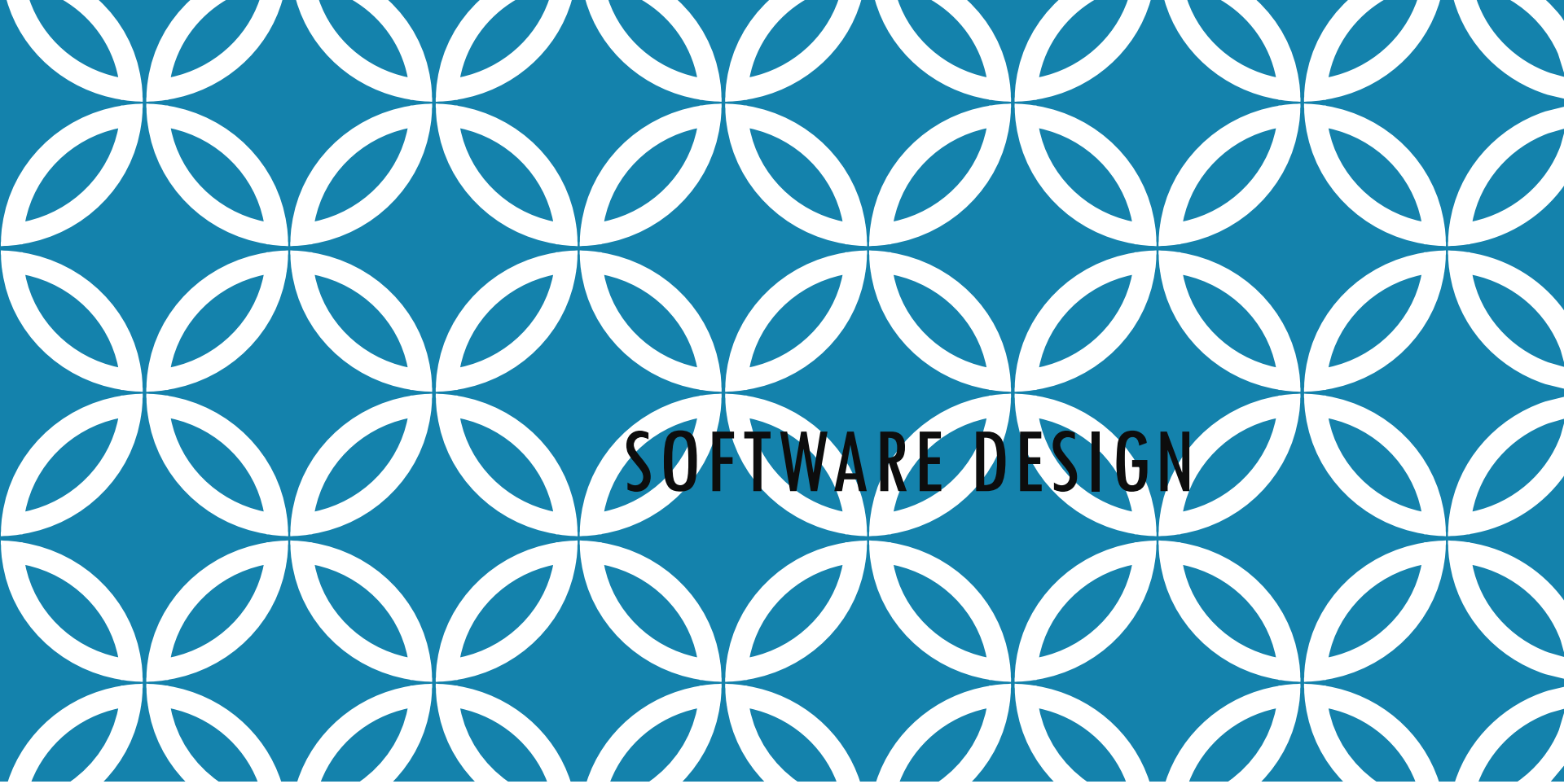


SOFTWARE DESIGN

Architectural Patterns 1

CONTENT

Architectural Patterns

- Layers
  - Client-Server
- Event-driven
  - Broker
  - Mediator
- MVC (and variants)

REFERENCES

Books

- Martin, Robert C., *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017.
- Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- F. Buschmann et. al, PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A System of Patterns, Wiley&Sons, 2001.[POSA]
- Artem Syromiatnikov, A Journey Through the Land of Model-View-* Design Patterns, MSc Thesis, 2014.
- Reid Holmes, MVC/MCP, Univ. of Waterloo course materials

Online resources

- O. Shelest, MVC, MVP, MVVM design patterns,
- https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/
- http://www.codeproject.com/KB/architecture/MVC_MVP_MVVM_design.aspx
- http://www.tinmegali.com/en/model-view-presenter-mvp-in-android-part-2/
- https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/
- https://www.infoq.com/articles/no-more-mvc-frameworks?utm_source=infoq&utm_campaign=user_page&utm_medium=link

LAST TIME

GRASP

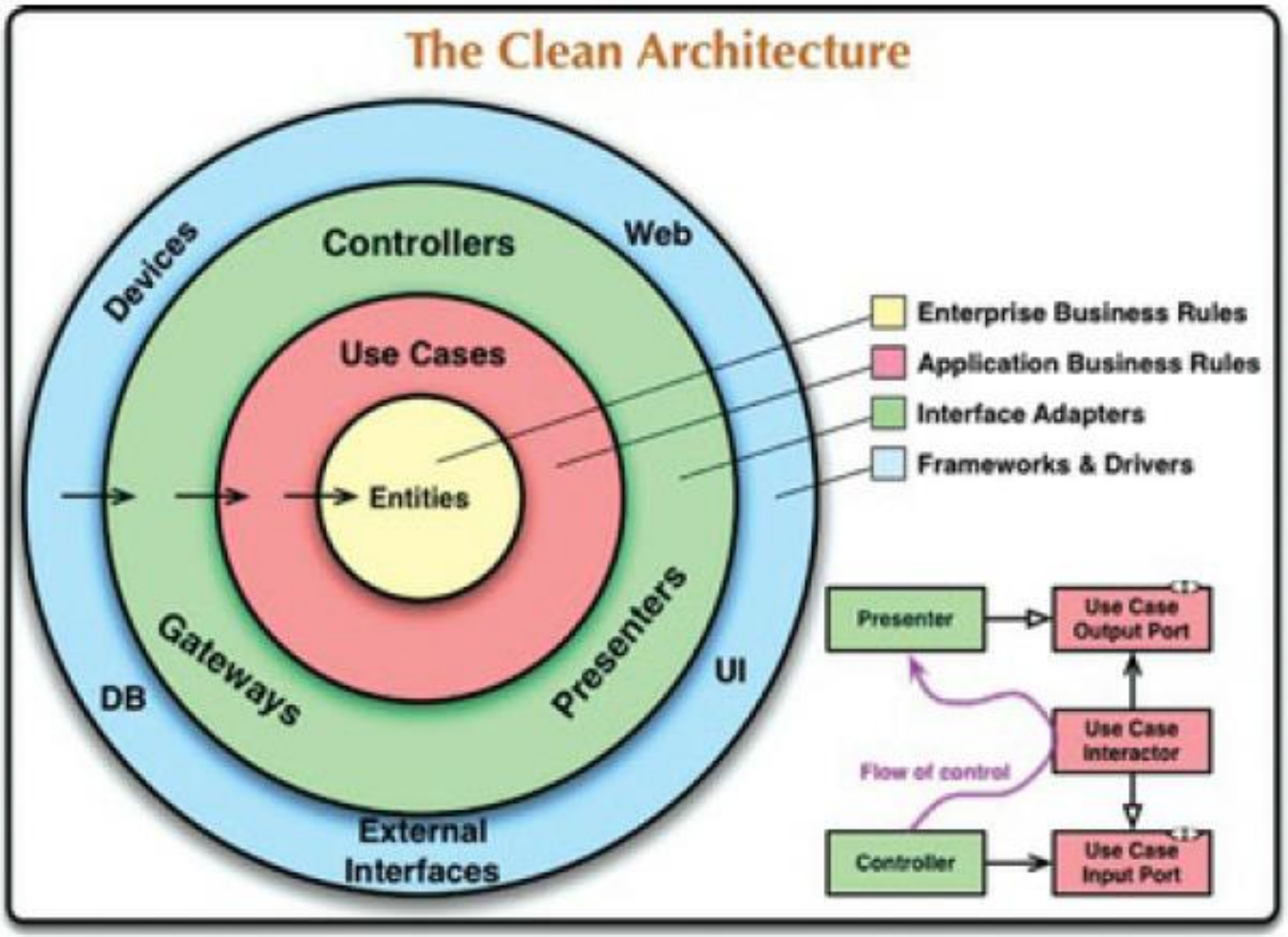
Package design principles

- Cohesion
- Coupling

CLEAN ARCHITECTURE

- *Independent of frameworks*. Frameworks are tools, they do not drive your architecture.
- *Testable*. The business rules can be tested without the UI, database, web server, or any other external element.
- *Independent of the UI*. The UI can change easily, without changing the rest of the system.
- *Independent of the database*. You can swap out Oracle or SQL Server for Mongo, BigTable, CouchDB, or something else.
- *Independent of any external agency*. In fact, your business rules don't know anything at all about the interfaces to the outside world.

DEPENDENCIES IN A CLEAN ARCHITECTURE



ARCHITECTURAL PATTERNS

Definition

- An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

[POSA, vol.1]

Different books present different taxonomies of patterns

STRUCTURAL AP

From Mud to Structure

Direct mapping Requirements -> Architecture?

Problems: non-functional qualities like

- availability
- reliability
- maintainability
- understandability…

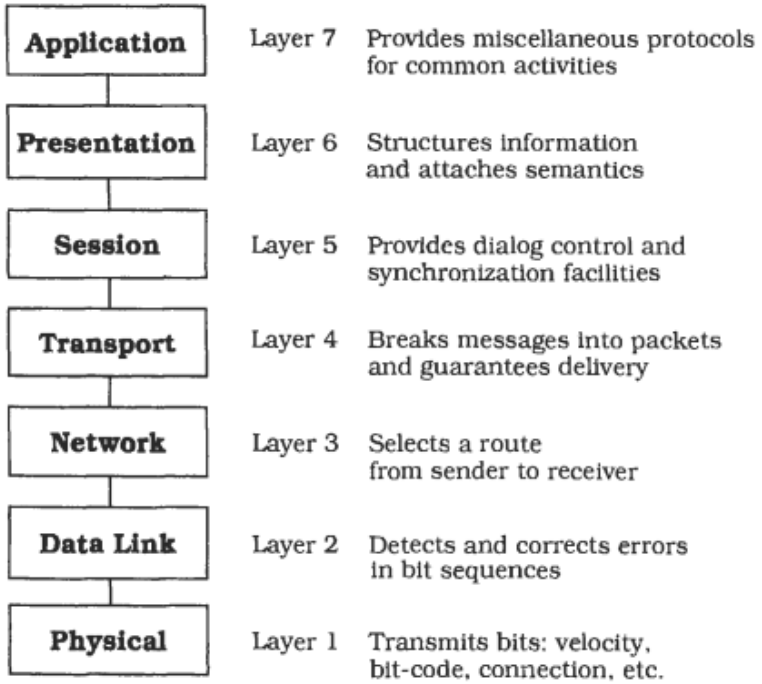
LAYERS

Definition

- The **Layers** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Example

- Networking: OSI 7-Layer Model



DESIGNING LAYERS

Addressed problems

- High-level and low-level operations
- High-level operations rely on low-level ones

$\Rightarrow$Vertical structuring

- Several operations on the same level on abstraction but highly independent

$\Rightarrow$Horizontal structuring

CONSTRAINTS

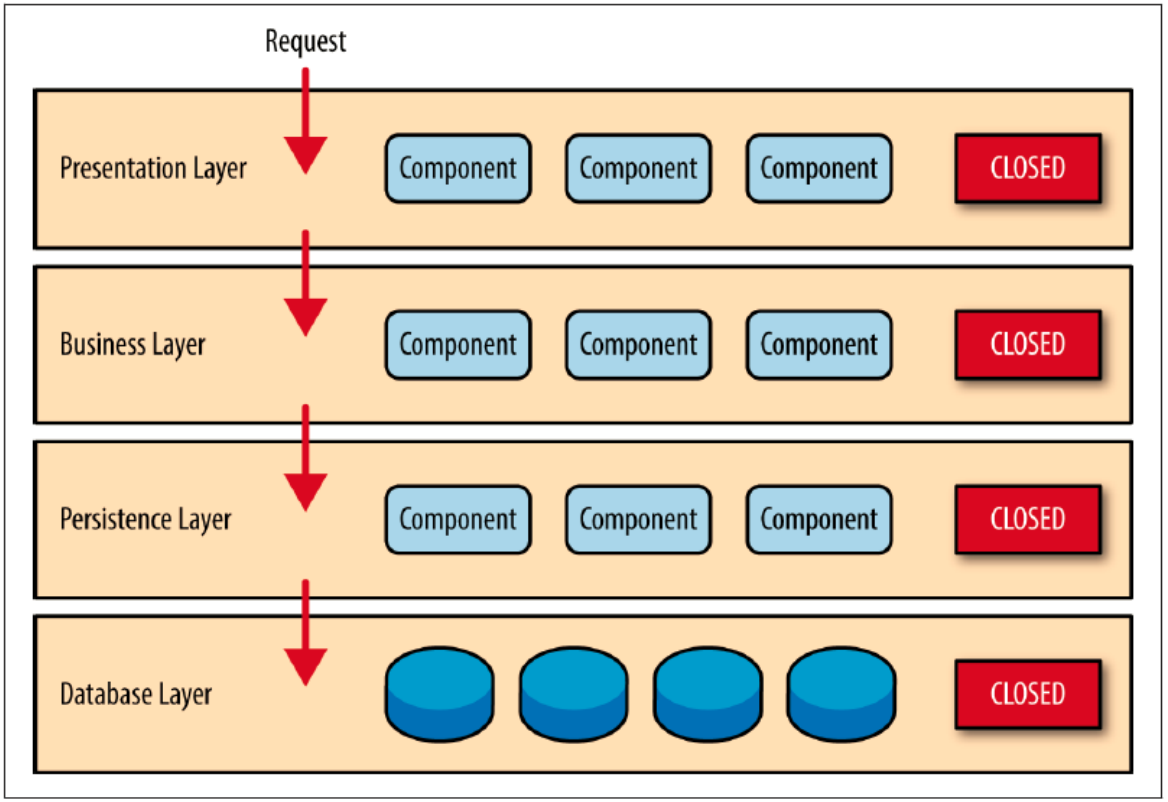
- Late source code/design **changes should not ripple through the system.**
- **Interfaces should be stable** and may even be prescribed by a standards body.
- **Parts** of the system **should be exchangeable.**
- It may be necessary to **build other systems** at a later date **with the same low-level issues** as the system you are currently designing.

CONSTRAINTS [2]

- **Similar responsibilities should be grouped** to help understandability and maintainability.
- There is **no 'standard' component granularity.**
- Complex components need further **decomposition.**
- The system will be built by a team of programmers, and work has to be subdivided along clear boundaries.

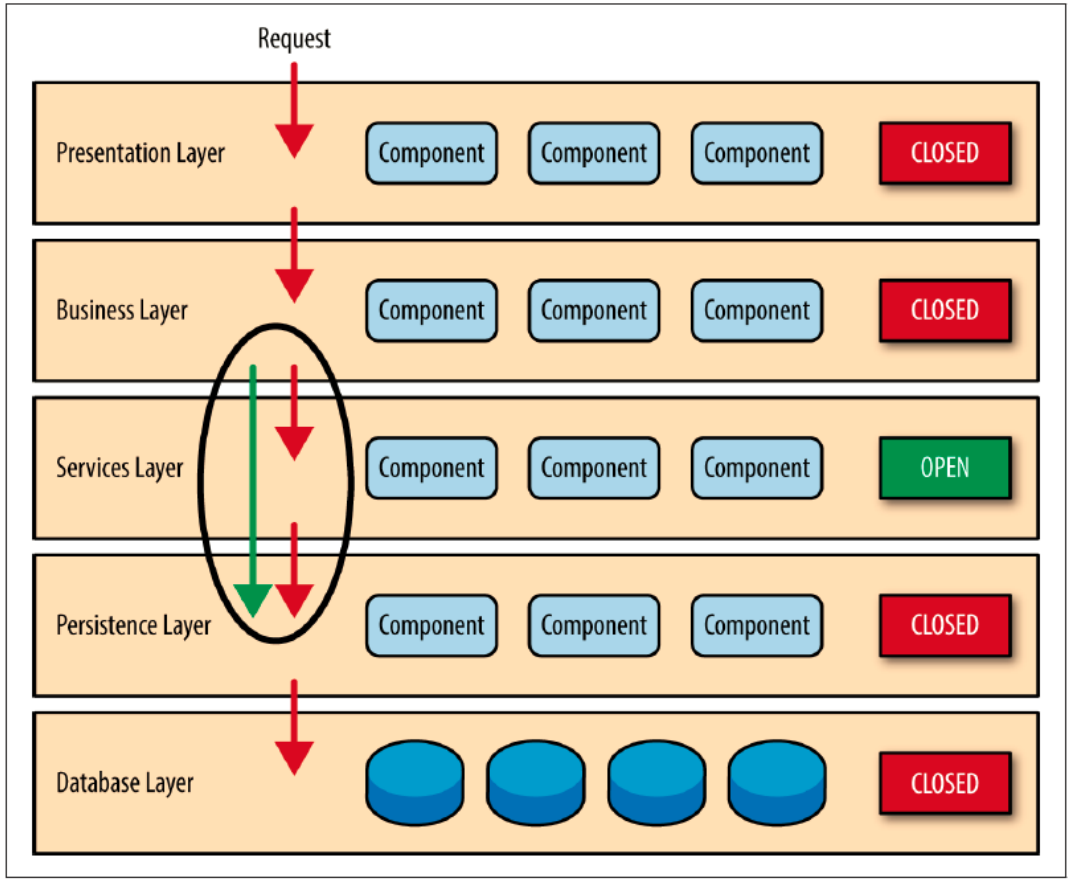
KEY CONCEPTS

- Closed layers
- Layers of isolation: changes made in one layer of the architecture don't impact components in other layers



VARIANTS

- **open layered system** (any component calls any other component)
- **semi-closed/semi-open architecture** allows calling more than one layer down



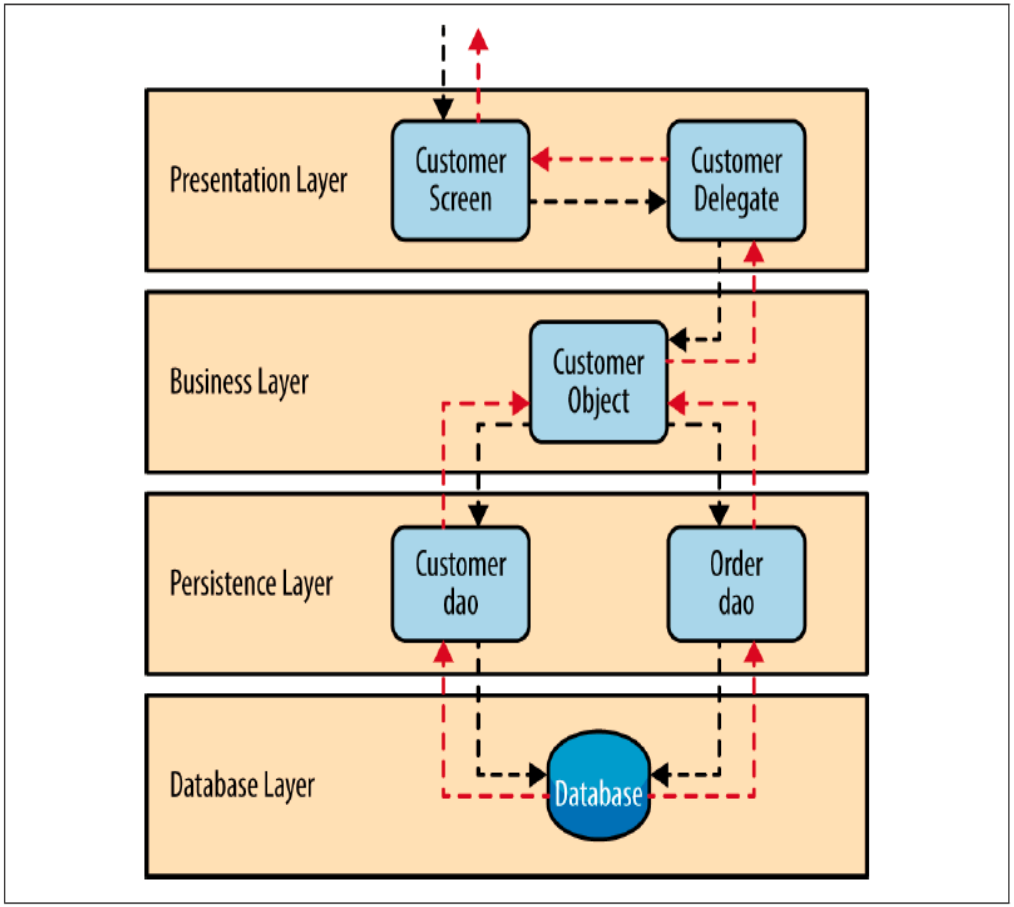
EXAMPLE

Customer Screen:
- Java Server Faces
- ASP (MS)

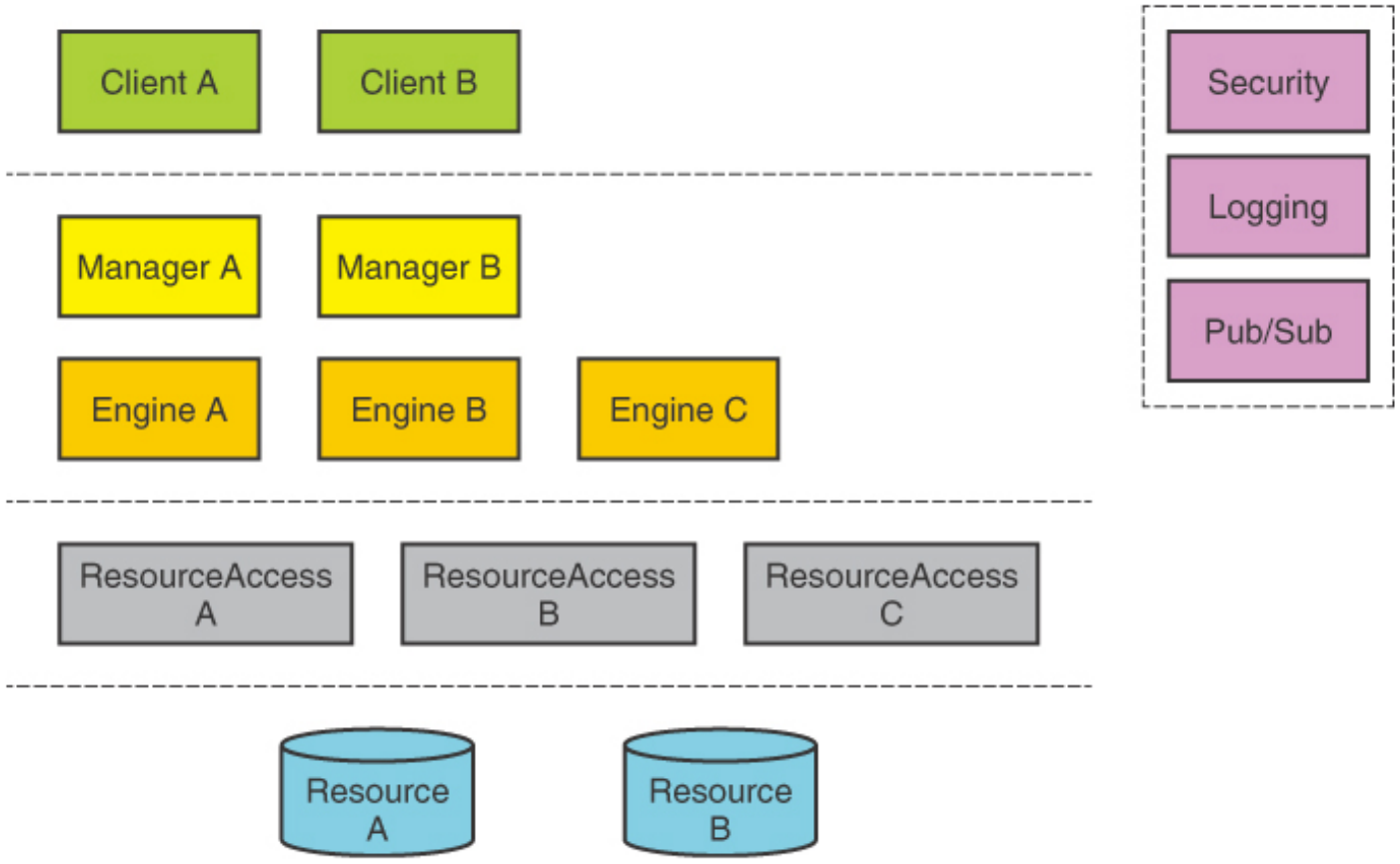
Customer Delegate:
- managed bean component

Customer Object:
- Local Spring Bean
- Remote EJB3 component
- C# (MS)

DAOs:
- POJOs
- MyBatis XML Mapper files
- Objects encapsulating raw JDBC call or Hibernate queries
- ADO (MS)



RELAXING THE RULES

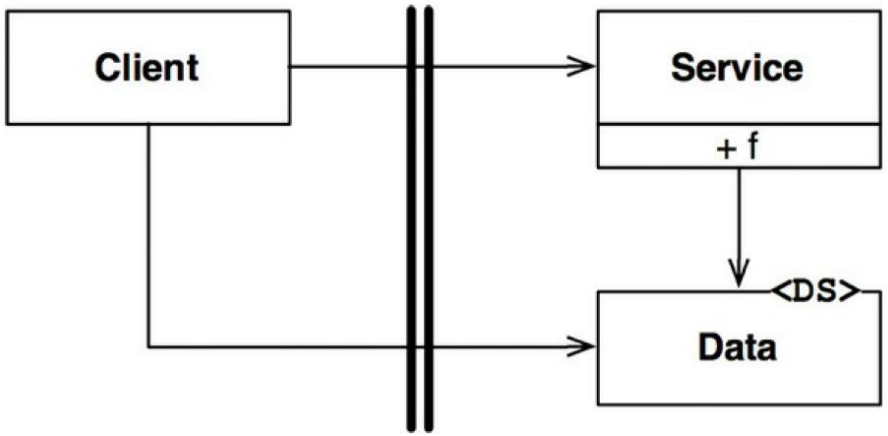


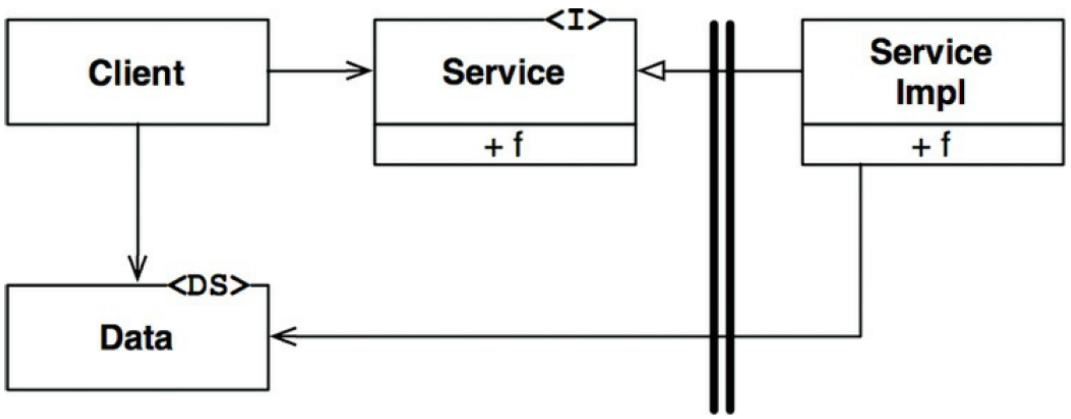
Righting software: A method for system and project design, Juval Lowy, O'Reilly 2020

DESIGN STEPS

- Define abstraction criterion for grouping tasks into layers
- Determine number of abstraction levels
- Name the layers and assign tasks to each of them
- Specify services (from a layer to another)
- Refine layering (iterate steps above)
- Specify interface for each layer
- Structure individual layers
- Specify communication between adjacent layers
- Decouple adjacent layers (callbacks for bottom-up)
- Design error-handling strategy

DEPENDENCY MANAGEMENT



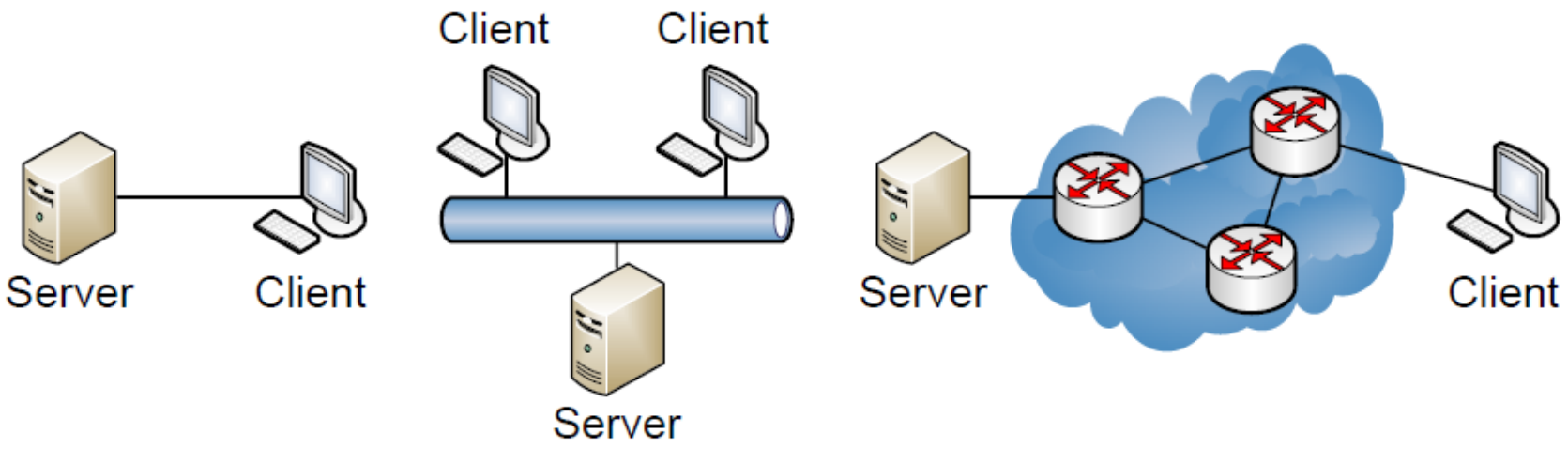


CLIENT-SERVER

Partition tasks and workload between provider of a resource (server) and requester (client)

Client does not share resources

Server hosts one or more server programs



STRUCTURE

Server

- Provides function or service (E.g. web server, web page, file server)
- Can service multiple clients

Client

- Consumes services
- Interacts with server to retrieve data
- Must understand response based on application protocol

Both can process data

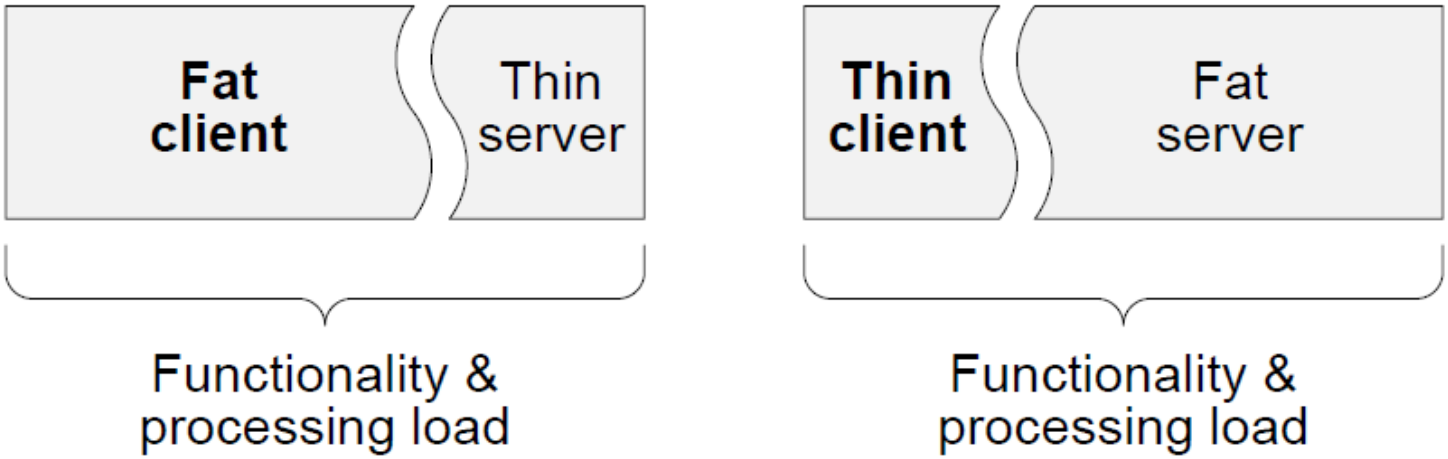
TYPES OF SERVERS AND CLIENTS

Servers

- Iterative (UDP-based servers, ex. Internet services like echo, daytime)
- Concurrent (TCP-based servers, ex. HTTP, FTP)
  - Thread-per-client
  - Thread pool

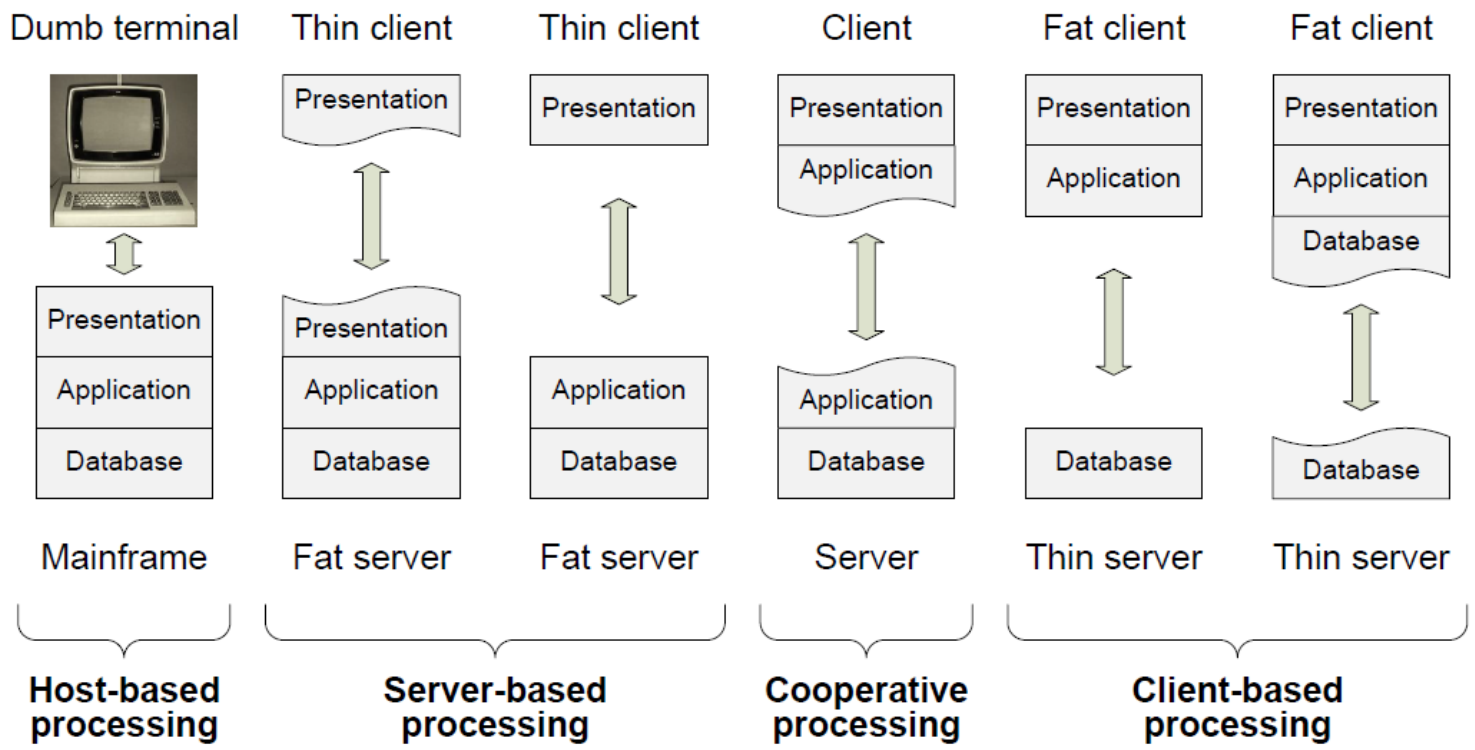
Clients

- Thin
- Fat



LAYERS VS. TIERS

- Layers – logical (ex. presentation, business logic, data access)
- Tiers – physical

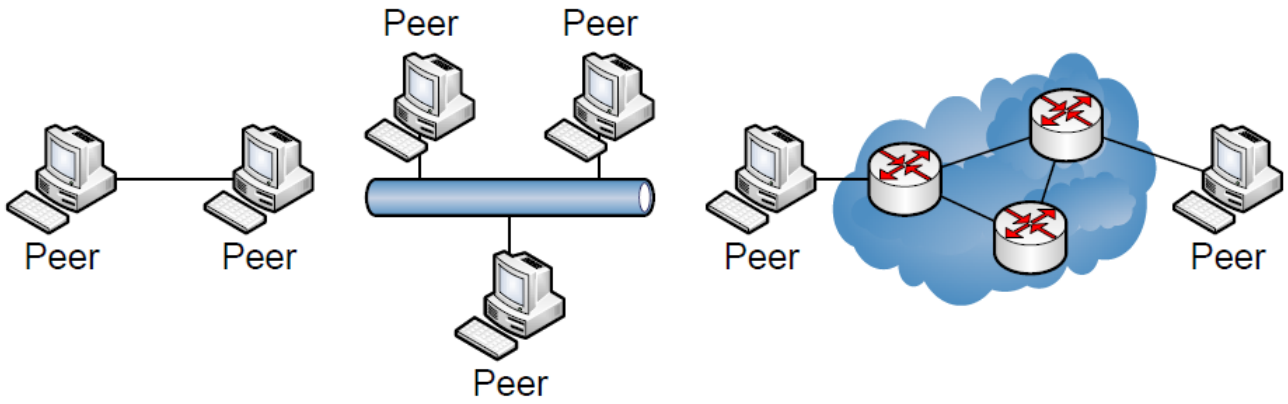


CONSEQUENCES ON QUALITY ATTRIBUTES

Quality Attribute	Issues
Availability	Servers in each tier can be replicated, so that if one fails, others remain available. Overall the application will provide a lower quality of service until the failed server is restored.
Failure handling	If a client is communicating with a server that fails, most web and application servers implement transparent failover. This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request.
Modifiability	Separation of concerns enhances modifiability, as the presentation, business and data management logic are all clearly encapsulated. Each can have its internal logic modified in many cases without changes rippling into other tiers.
Performance	This architecture has proven high performance. Key issues to consider are the amount of concurrent threads supported in each server, the speed of connections between tiers and the amount of data that is transferred. As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request.
Scalability	As servers in each tier can be replicated, and multiple server instances run on the same or different servers, the architecture scales out and up well. In practice, the data management tier often becomes a bottleneck on the capacity of a system.

PEER-TO-PEER

- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages



PEER-TO-PEER [2]

- Topology: Network (may have redundant connections between peers) can **vary arbitrarily and dynamically**
- Supports **decentralized** computing with flow of control and resources distributed among peers.
- Highly **robust** in the face of failure of any given node.
- **Scalable** in terms of access to resources and computing power.
- **Drawbacks:**
  - Poor security
  - Nodes with shared resources have poor performance

EVENT-DRIVEN (DISTRIBUTED) ARCHITECTURES

- (Distributed) **asynchronous** architecture pattern
- Highly **scalable** applications
- Highly **adaptable** by integrating highly decoupled, single-purpose event processing components that asynchronously receive and process events

- 2 main topologies
  - Broker
  - Mediator

BROKER

Definition

- The Broker architectural pattern can be used to structure distributed software systems with **decoupled components** that interact by remote service invocations. A broker component is responsible for coordinating communication.

Example

- SOA

BROKER

Context

- The environment is a **distributed** and possibly **heterogeneous** system with **independent, cooperating components.**

Problems

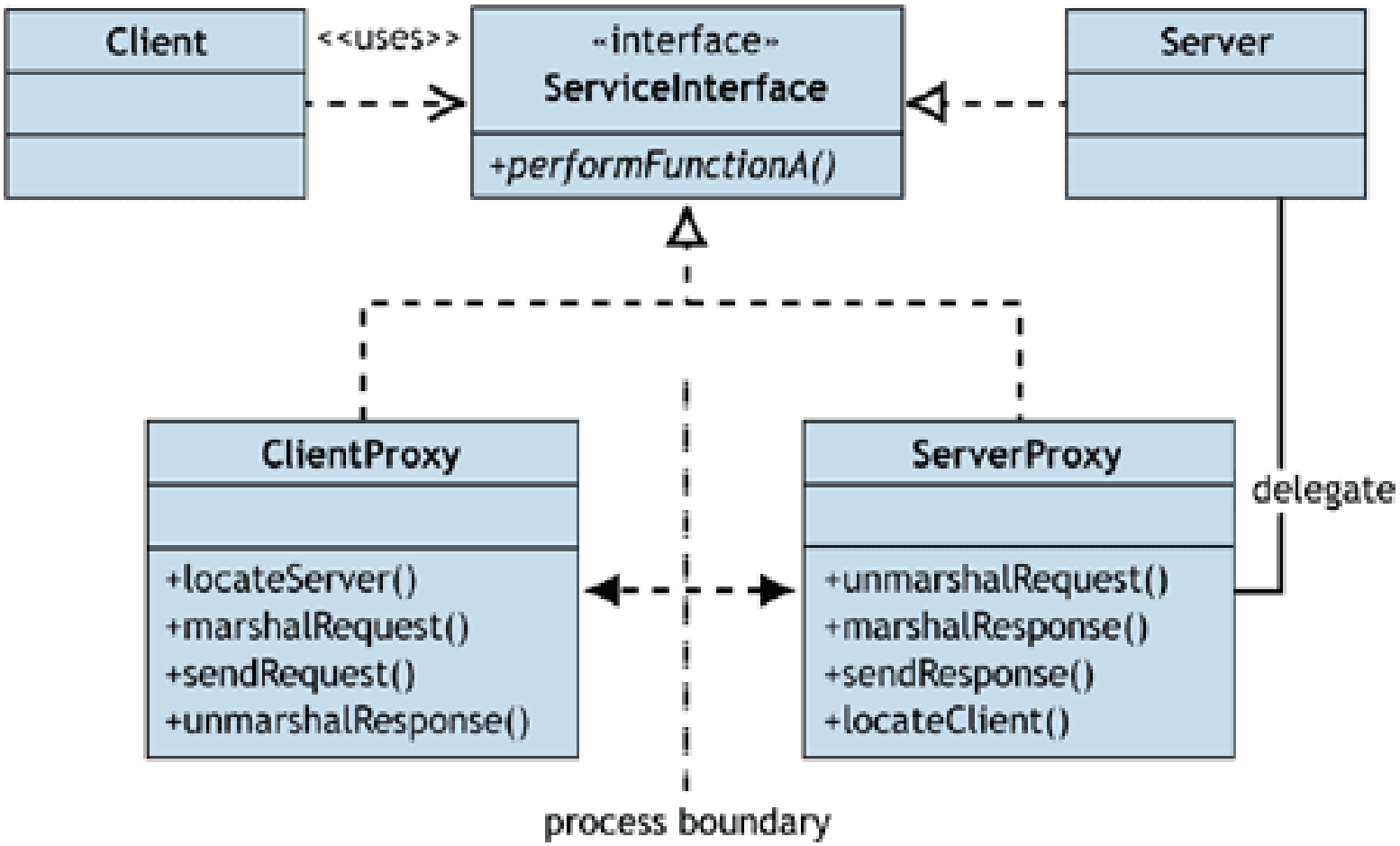
- System = set of decoupled and inter-operating components
- Inter-process communication
- Services for adding, removing, exchanging, activating and locating components are also needed.

CONSTRAINTS

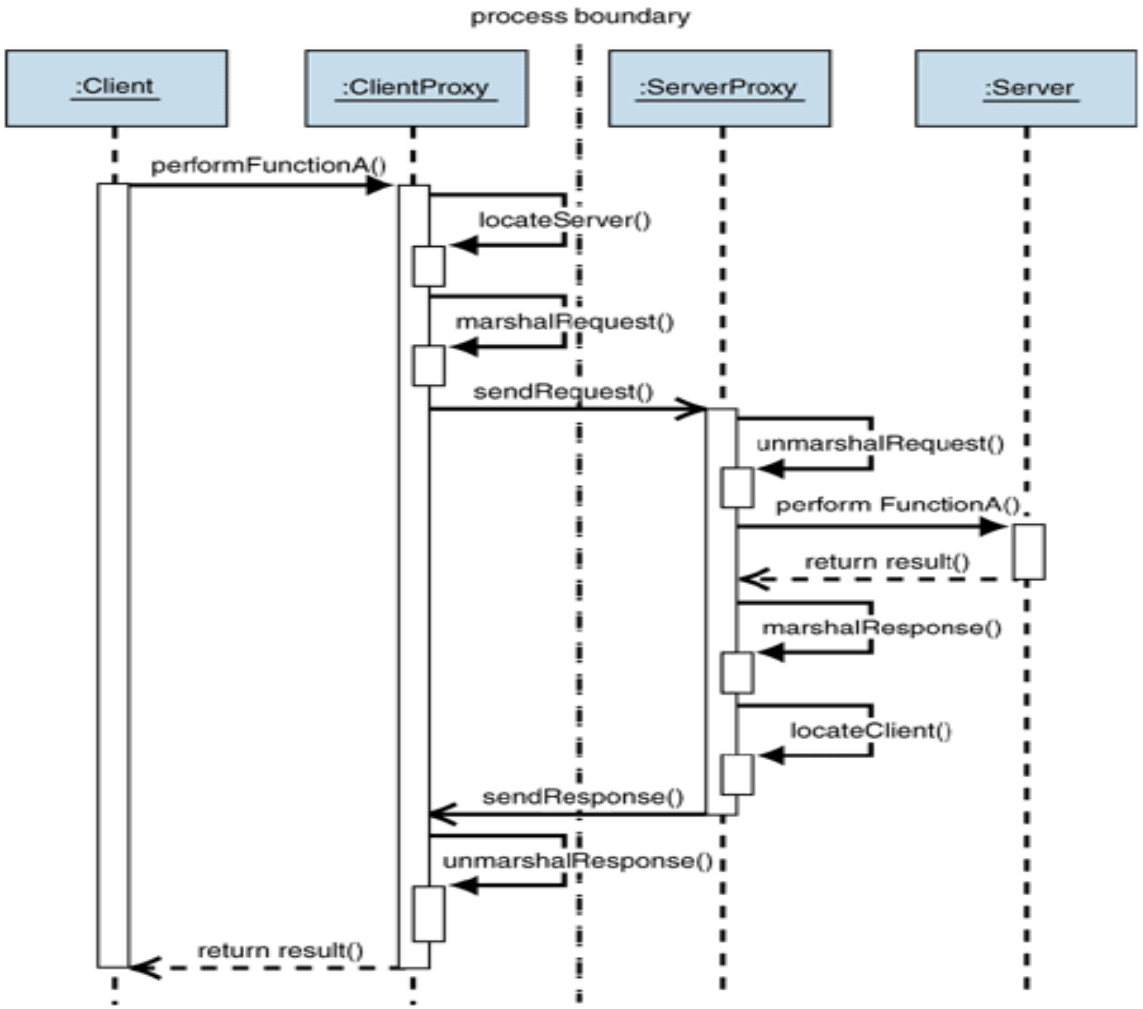
- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

NON-DISTRIBUTED SYSTEM [MSDN]

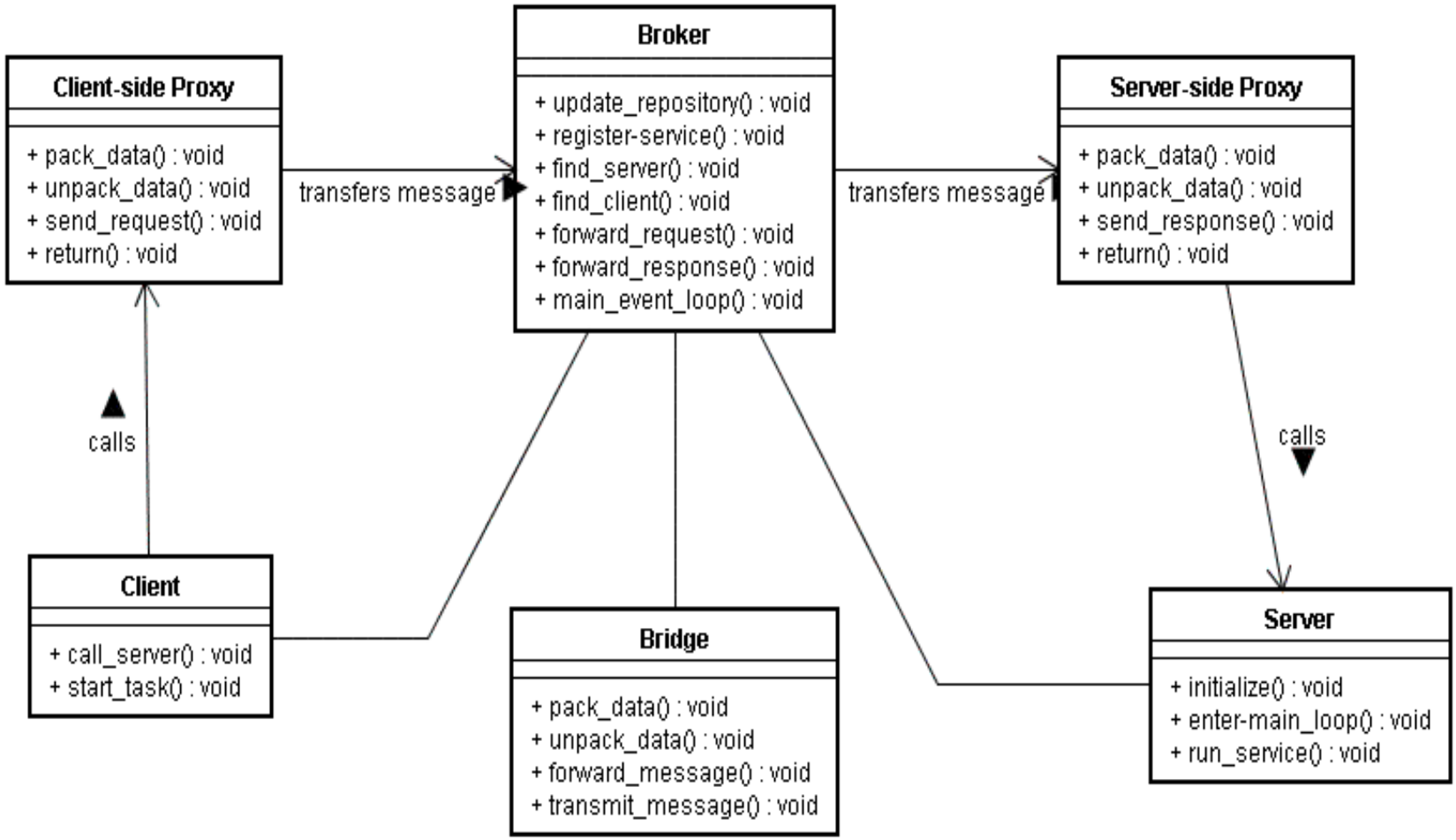
DISTRIBUTED SYSTEM [MSDN]



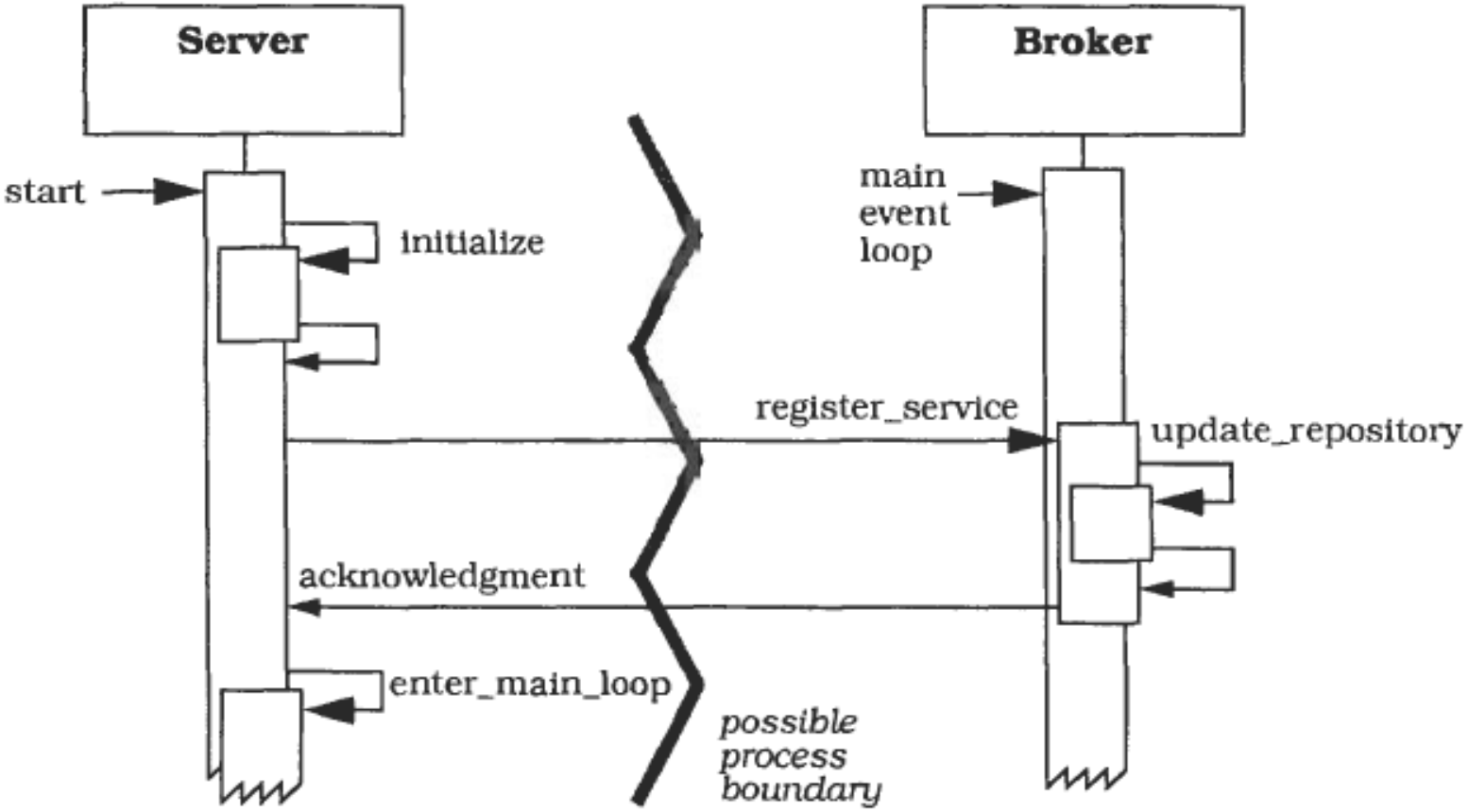
DISTRIBUTED SYSTEM – PROBLEMS?



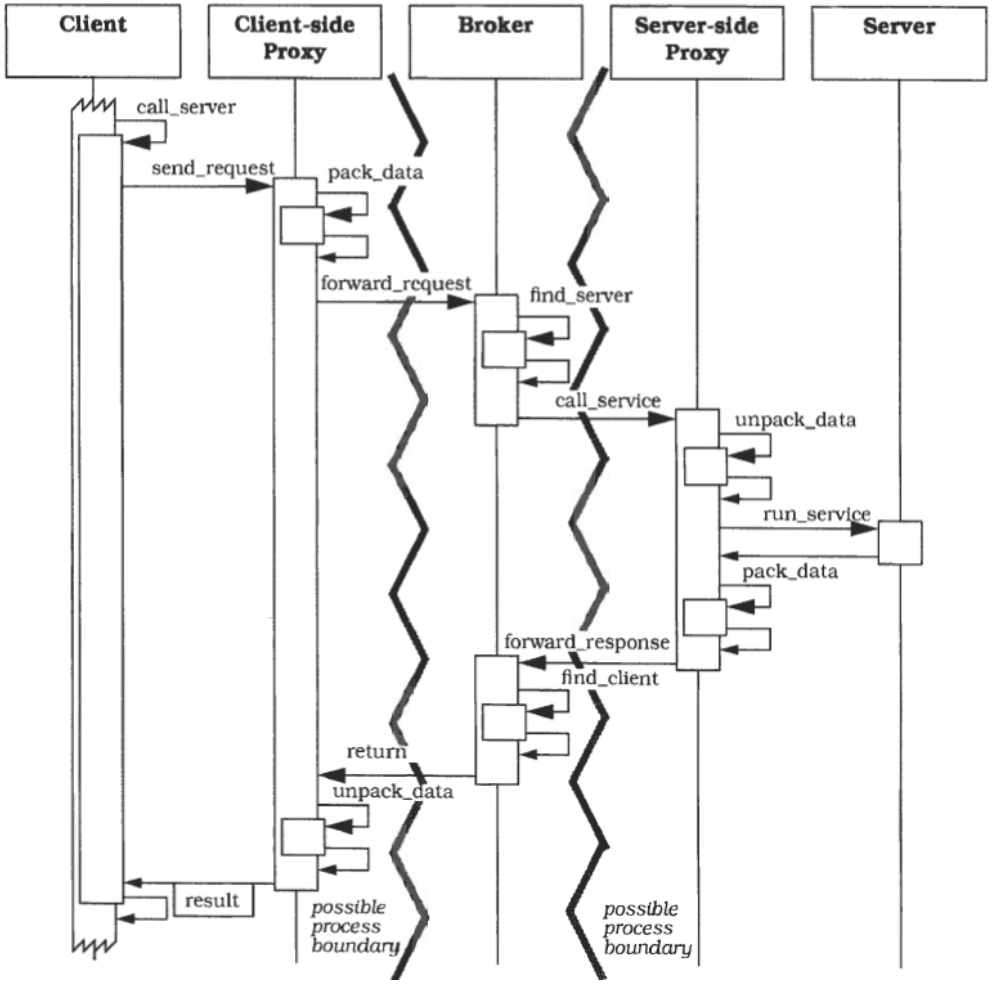
SOLUTION


Broker
+ update_repository() : void
+ register-service() : void
+ find_server() : void
+ find_client() : void
+ forward_request() : void
+ forward_response() : void
+ main_event_loop() : void
Client-side Proxy
+ pack_data() : void
+ unpack_data() : void
+ send_request() : void
+ return() : void
transfers message
Server-side Proxy
+ pack_data() : void
+ unpack_data() : void
+ send_response() : void
+ return() : void
transfers message
calls
calls
Client
+ call_server() : void
+ start_task() : void
Bridge
+ pack_data() : void
+ unpack_data() : void
+ forward_message() : void
+ transmit_message() : void
Server
+ initialize() : void
+ enter-main_loop() : void
+ run_service() : void

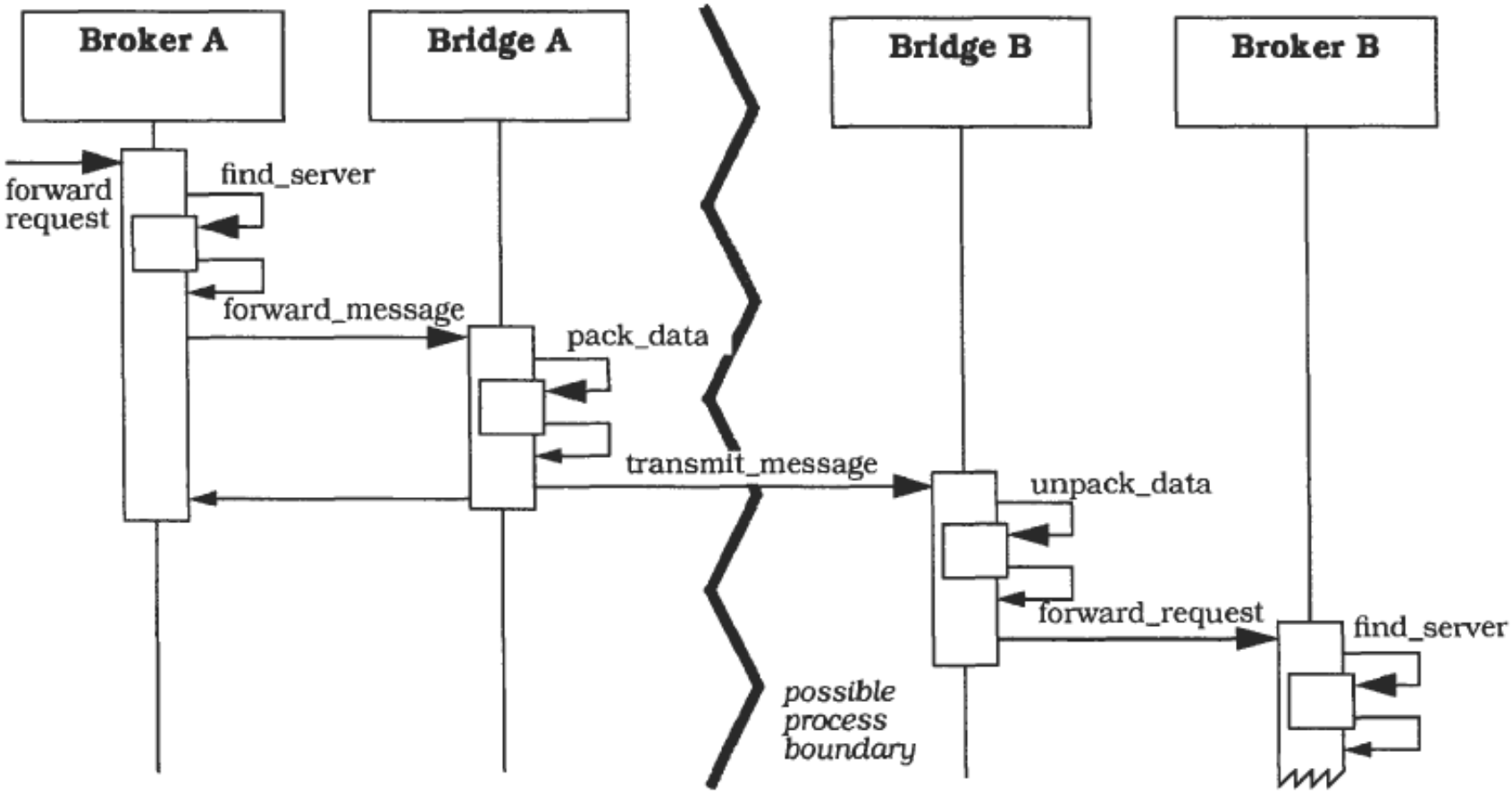
SCENARIO I – SERVER REGISTRATION



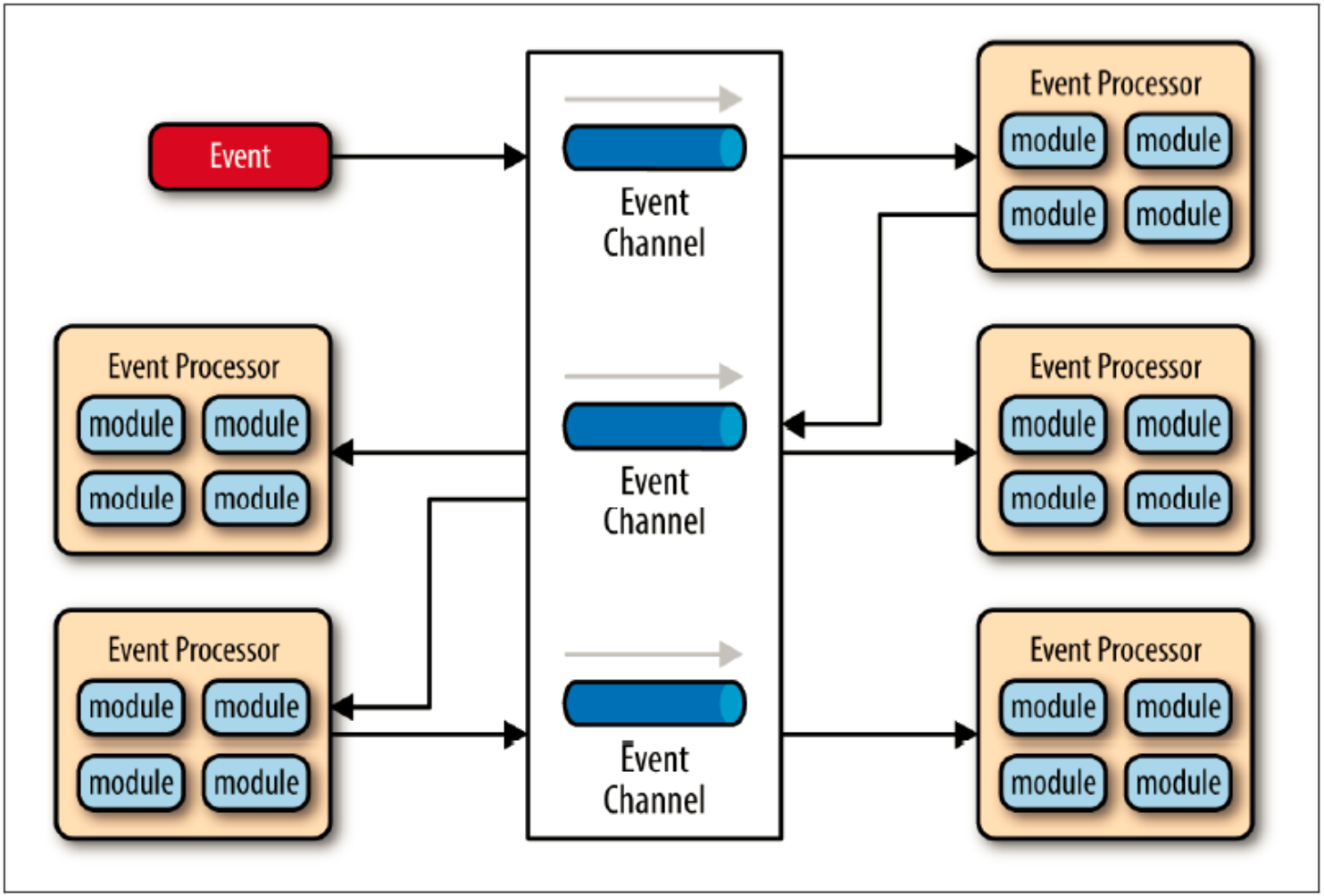
SCENARIO II – BROKER CONNECTING CLIENT AND SERVER



SCENARIO III – BRIDGE CONNECTING BROKERS



EVENT-DRIVEN PERSPECTIVE



CONSEQUENCES

Benefits

- Location transparency
- Changeability and extensibility of components
- Portability of a Broker System
- Interoperability between Broker Systems
- Reusability

Liabilities

- Reliability – remote process availability, lack of responsiveness, broker reconnection
- Lack of atomic transactions for a single business process

CONSEQUENCES ON QUALITY ATTRIBUTES

Quality Attribute	Issues
Availability	To build high availability architectures, brokers must be replicated. This is typically supported using similar mechanisms to messaging and publish-subscribe server clustering.
Failure handling	As brokers have typed input ports, they validate and discard any messages that are sent in the wrong format. With replicated brokers, senders can fail over to a live broker should one of the replicas fail.
Modifiability	Brokers separate the transformation and message routing logic from the senders and receivers. This enhances modifiability, as changes to transformation and routing logic can be made without affecting senders or receivers.
Performance	Brokers can potentially become a bottleneck, especially if they must service high message volumes and execute complex transformation logic. Their throughput is typically lower than simple messaging with reliable delivery.
Scalability	Clustering broker instances makes it possible to construct systems scale to handle high request loads.

MEDIATOR

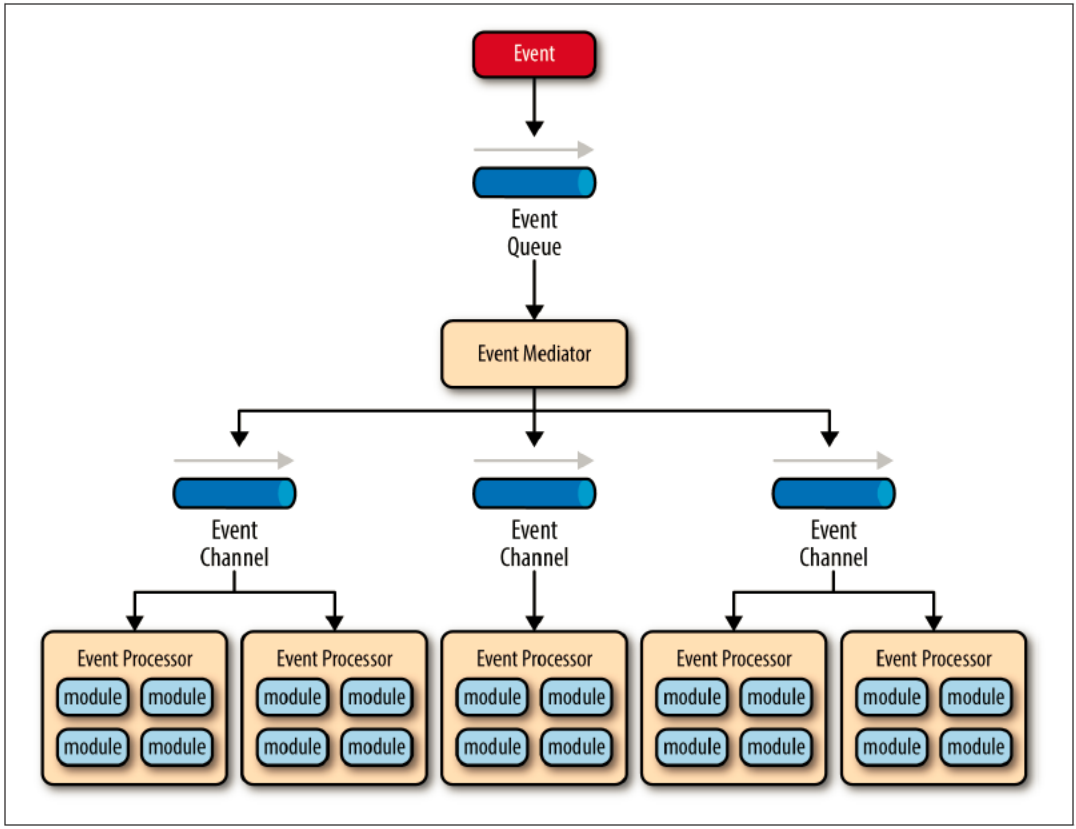
- for events that have multiple steps and require some level of orchestration to process the event

Event queue (hosts initial events)
- Message queue
- Web service endpoint

Events
- Initial
- Processing

Event channel (passes processing events)
- Message queue
- Message topic

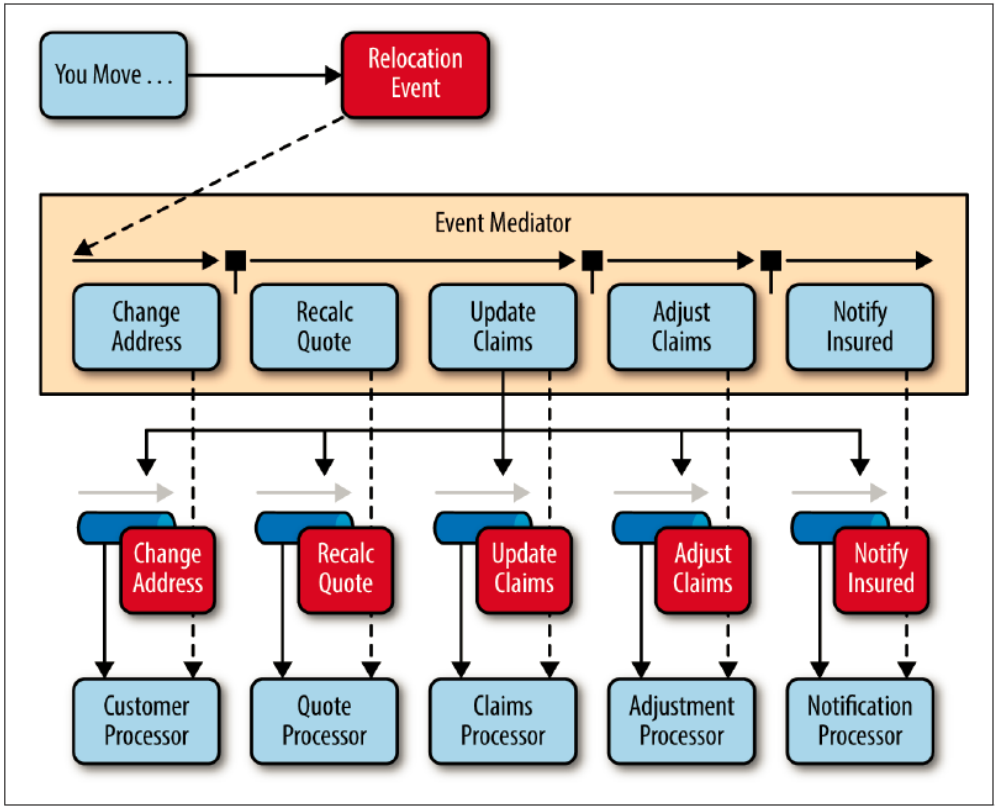
Event processor
- self-contained, independent, highly decoupled business logic components



EXAMPLE (INSURANCE APP)

Event mediator

- open source integration hubs such as Spring Integration, Apache Camel, or Mule ESB
- BPEL (business process execution language) coupled with a BPEL engine (ex. Apache ODE)
- business process manager (BPM) (ex. jBPM)



INTERACTIVE SYSTEMS

Context

- **Interactive** applications with a **flexible human-computer interface**.

Problem

- User interfaces are especially prone to change requests.
- Different users place conflicting requirements on the user interface.

MODEL VIEW CONTROLLER (MVC)

Solution

- 3 areas: handle input, processing, output

- The **Model** component encapsulates core data and functionality (processing).

- **View** components display information to the user. A view obtains the data from the model (output).

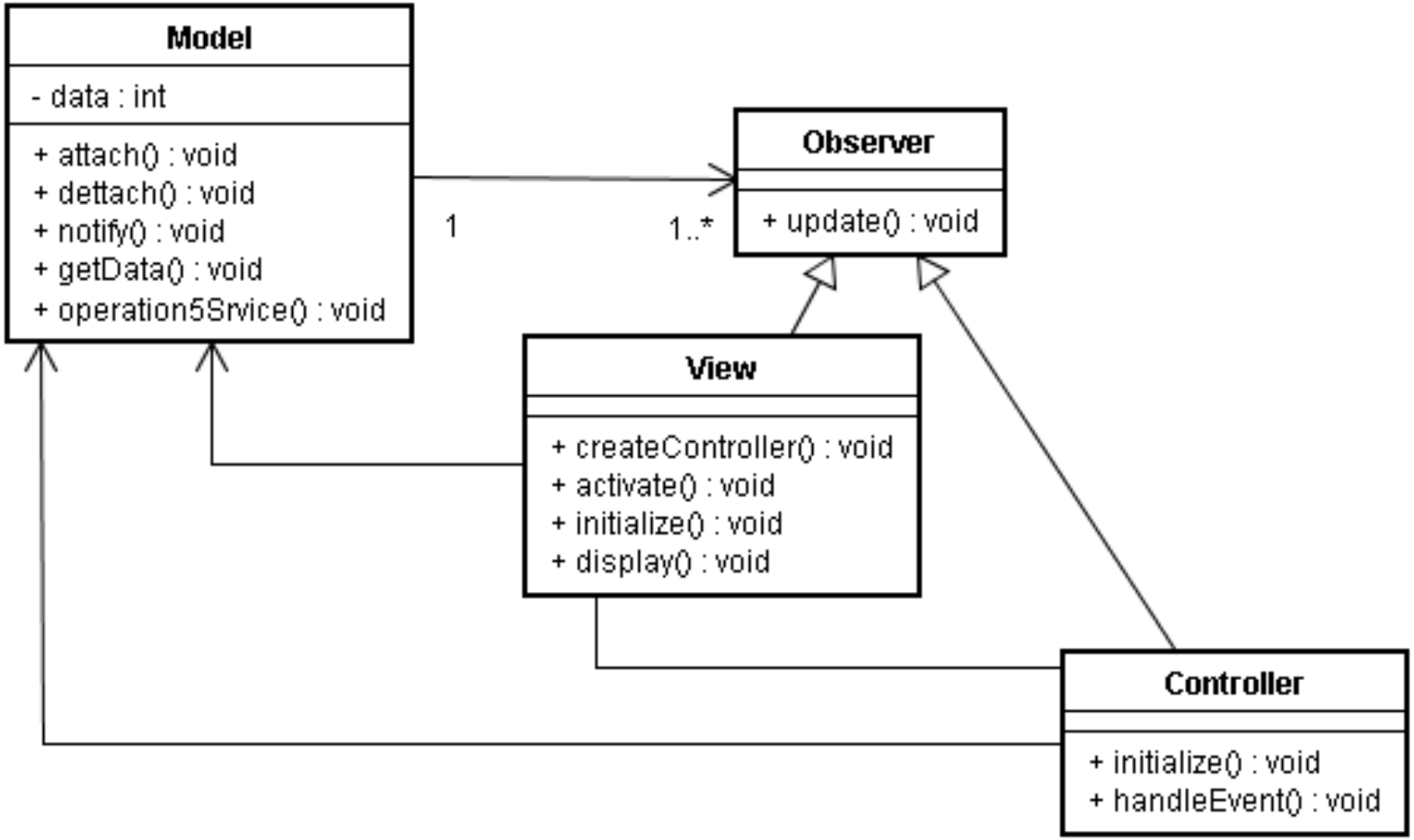
- Each view has an associated **Controller** component. Controllers handle input.

MVC

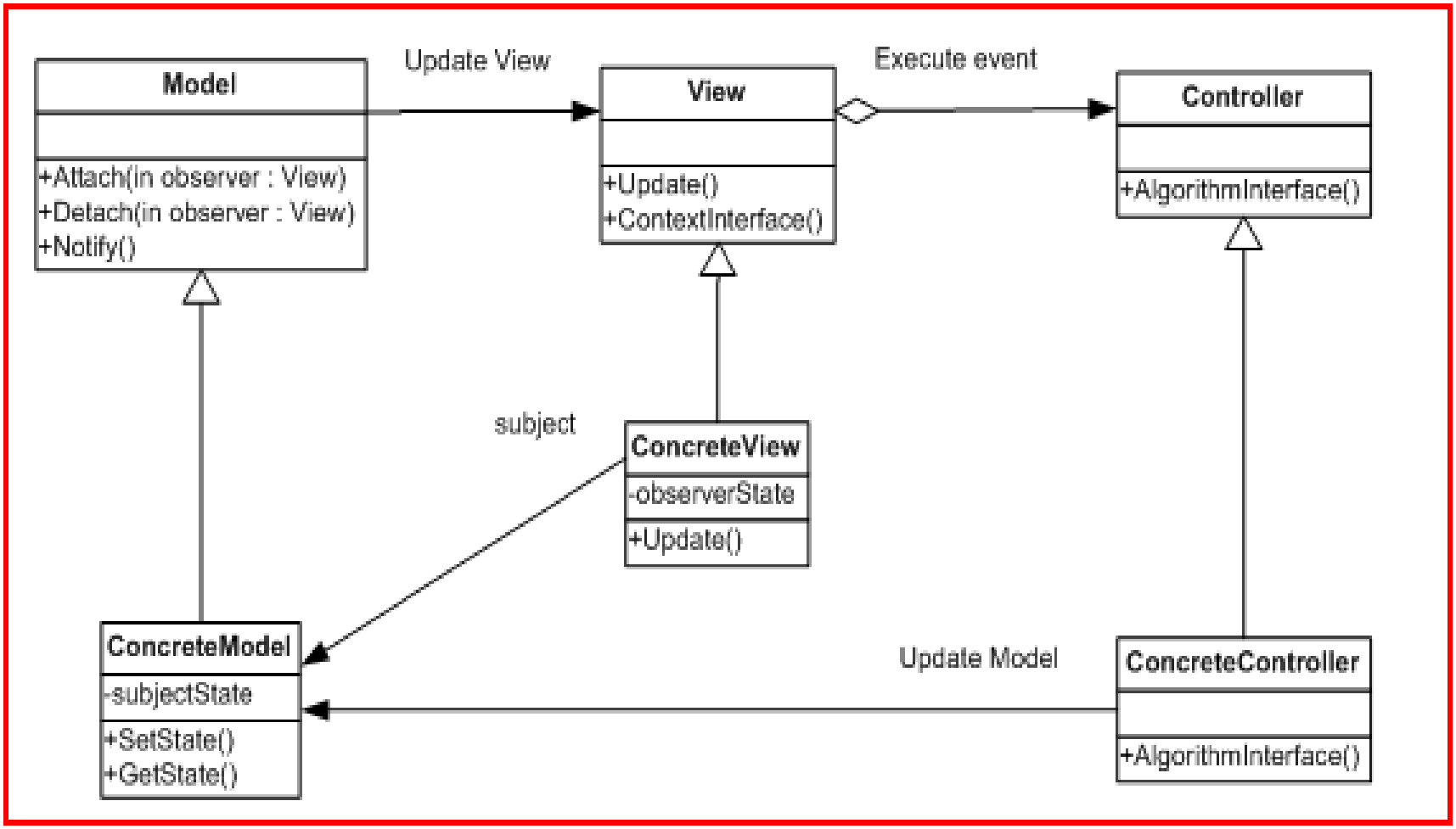
Constraints

- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behavior of the application must reflect data manipulations immediately.
- Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

MVC STRUCTURE



MVC MORE DETAILED



THE MODEL

- encapsulates and manipulates the **domain data** to be rendered

- has **no** idea how to **display** the information is has **nor does it interact** with the user or receive any user input

- encapsulates the functionality necessary to manipulate, obtain, and deliver that data to others, *independent* of any user interface or any user input device

THE VIEW

- Is **a specific visual rendering** of the information contained in the model (graphical, text-based).
- **Multiple views** may present multiple renditions of the data in the model
- Each view is ***dependent*** of a model
- When the model changes, all dependent views are updated

THE CONTROLLER

- handles user input. They "listen" for user direction, and *handle* requests using the model and views
- often watches mouse events and keyboard events
- allows the decoupling of the model and its views, allowing views to simply render data and models to simply encapsulate data
- is "paired up" with collections of view types, so that a "pie graph" view would be associated with its own "pie graph" controller, etc.
- **The *behavior* of the controller is dependent upon the *state* of the model**

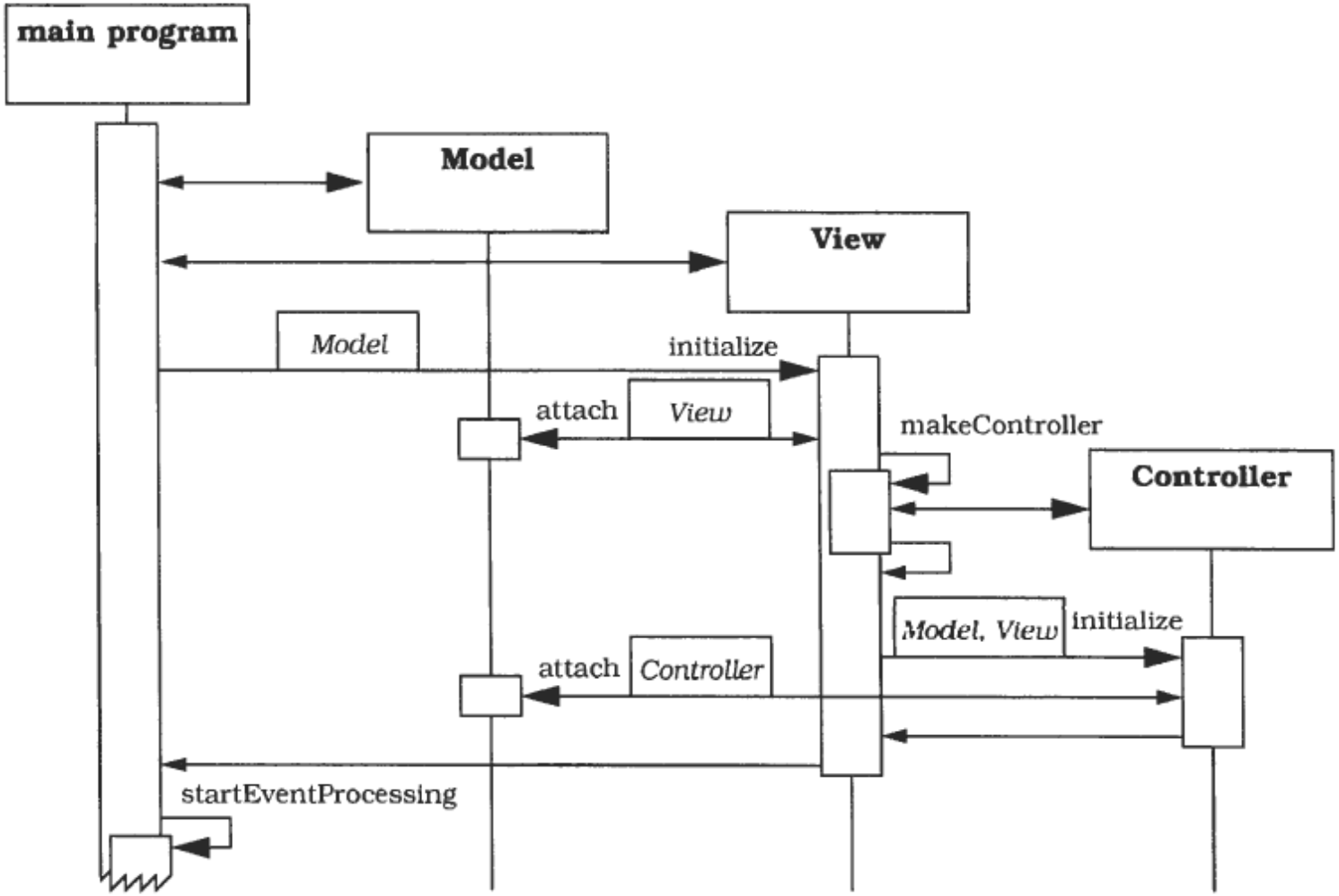
HOW DOES THIS WORK?

- The **model has a list of views** it supports
- Each **view has a reference to its model**, as well as **its supporting controller**
- Each **controller has a reference to the view** it controls, as well as **to the model** the view is based on. However, models know nothing about controllers.
- On user input, the controller notifies the model which in turn notifies its views of a change

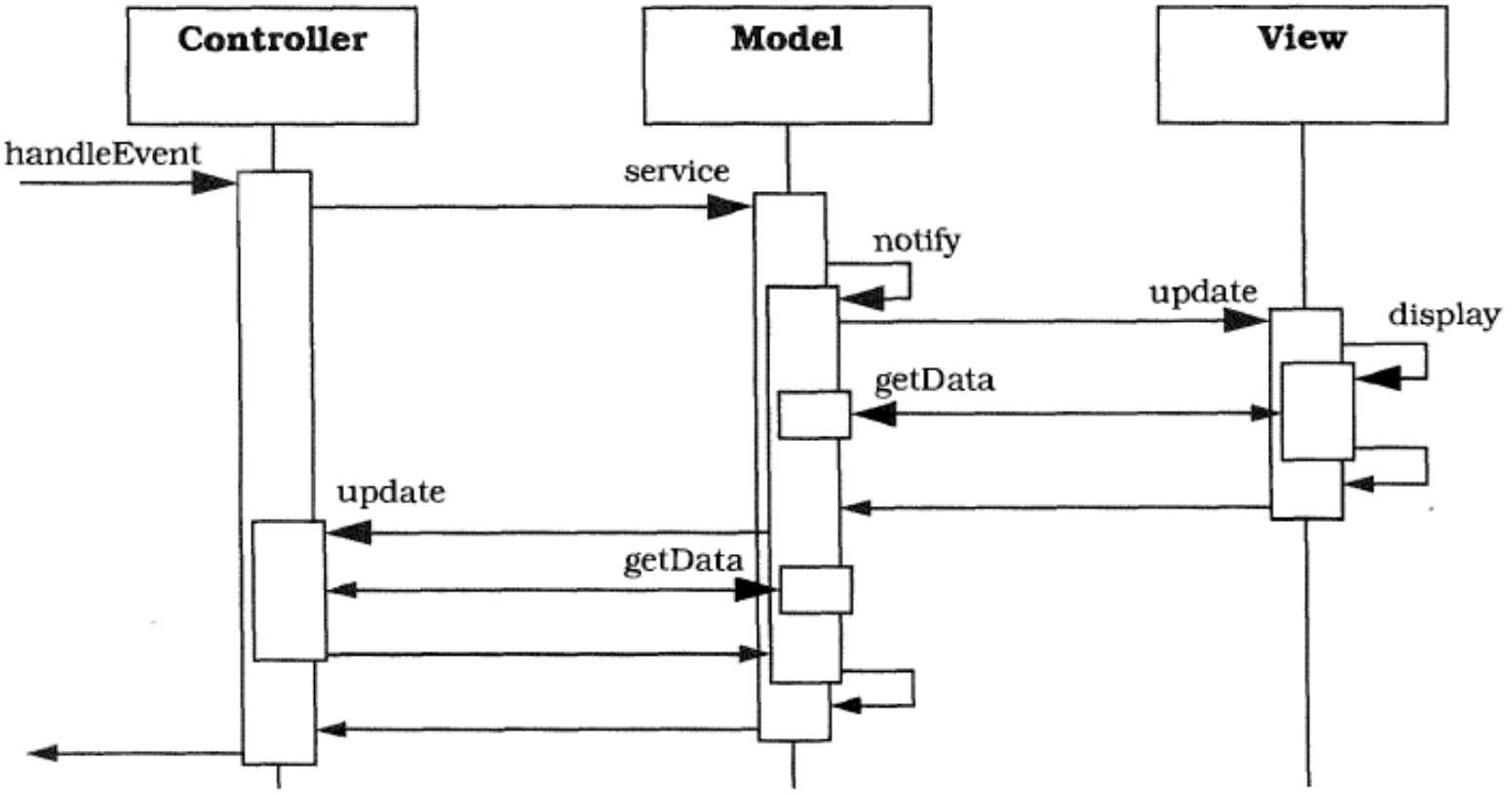
Changing a controller's view will give a different *look*.

Changing a view's controller will give a different *feel*.

SCENARIO I – CREATING THE M,V,C OBJECTS



SCENARIO II – EVENT HANDLING



CONSEQUENCES

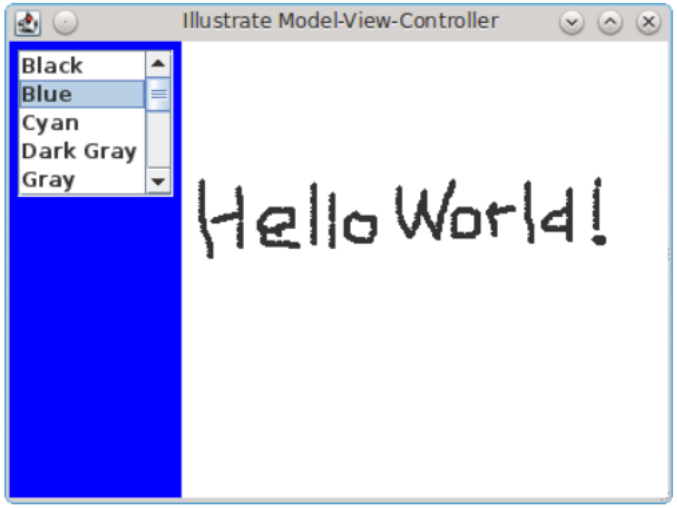
Benefits

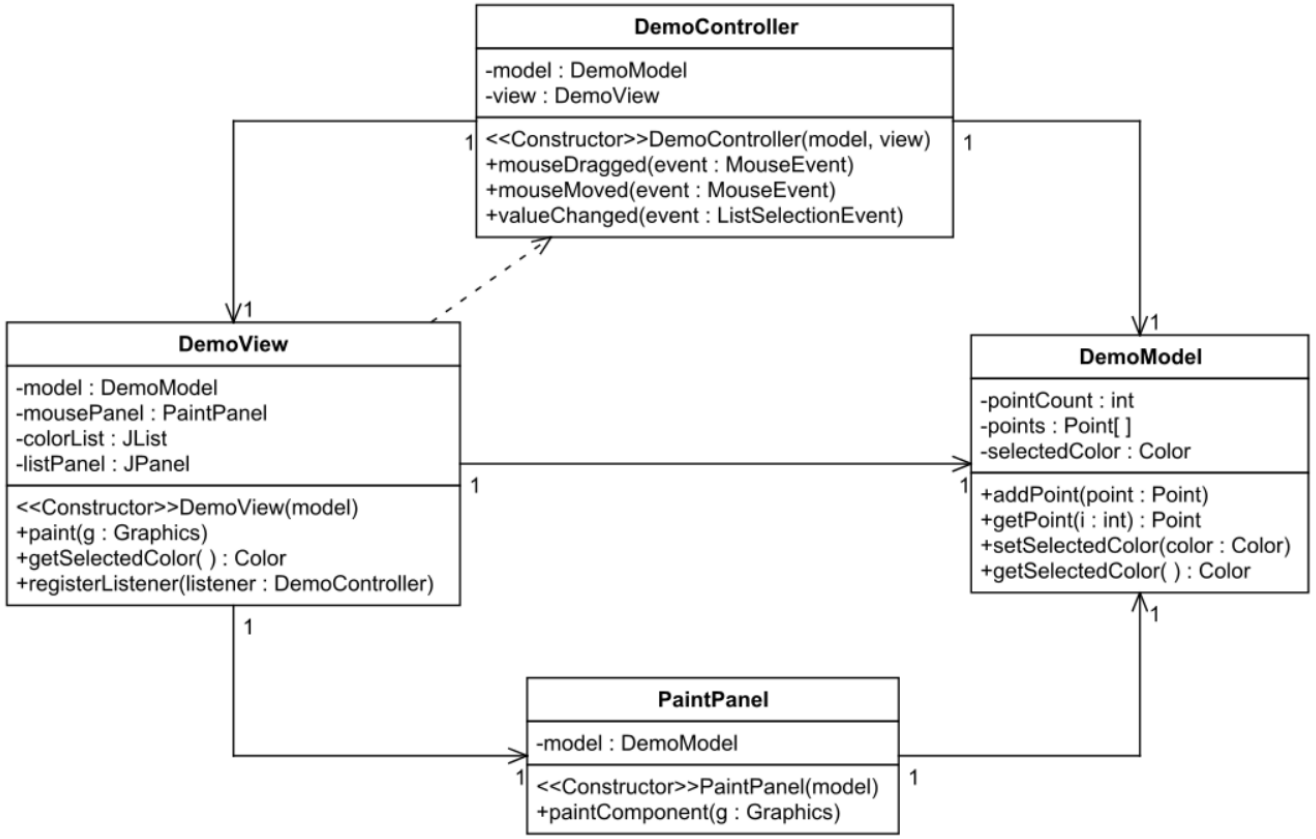
- Multiple views of the same model.
- Synchronized views.
- 'Pluggable' views and controllers
- Exchangeability of 'look and feel'.
- Framework potential.

Liabilities

- Increased complexity.
- Potential for excessive number of updates.
- Intimate connection between view and controller.
- Close coupling of views and controllers to a model
- Inefficiency of data access in view.
- Inevitability of change to view and controller when porting.

MVC EXAMPLE


Illustrate Model-View-Controller
Black
Blue
Cyan
Dark Gray
Gray
Hello World!


DemoController
-model : DemoModel
-view : DemoView
<<Constructor>>DemoController(model, view)
+mouseDragged(event : MouseEvent)
+mouseMoved(event : MouseEvent)
+valueChanged(event : ListSelectionEvent)
DemoView
-model : DemoModel
-mousePanel : PaintPanel
-colorList : JList
-listPanel : JPanel
<<Constructor>>DemoView(model)
+paint(g : Graphics)
+getSelectedColor( ) : Color
+registerListener(listener : DemoController)
DemoModel
-pointCount : int
-points : Point[ ]
-selectedColor : Color
+addPoint(point : Point)
+getPoint(i : int) : Point
+setSelectedColor(color : Color)
+getSelectedColor( ) : Color
PaintPanel
-model : DemoModel
<<Constructor>>PaintPanel(model)
+paintComponent(g : Graphics)
1

MODEL VIEW PRESENTER

Intent

- Separation between
  - Data
  - Business Logic
  - UI
- Enforce single responsibility: M-V-P
- Reduce coupling
- Facilitate isolated testing

MVP STRUCTURE

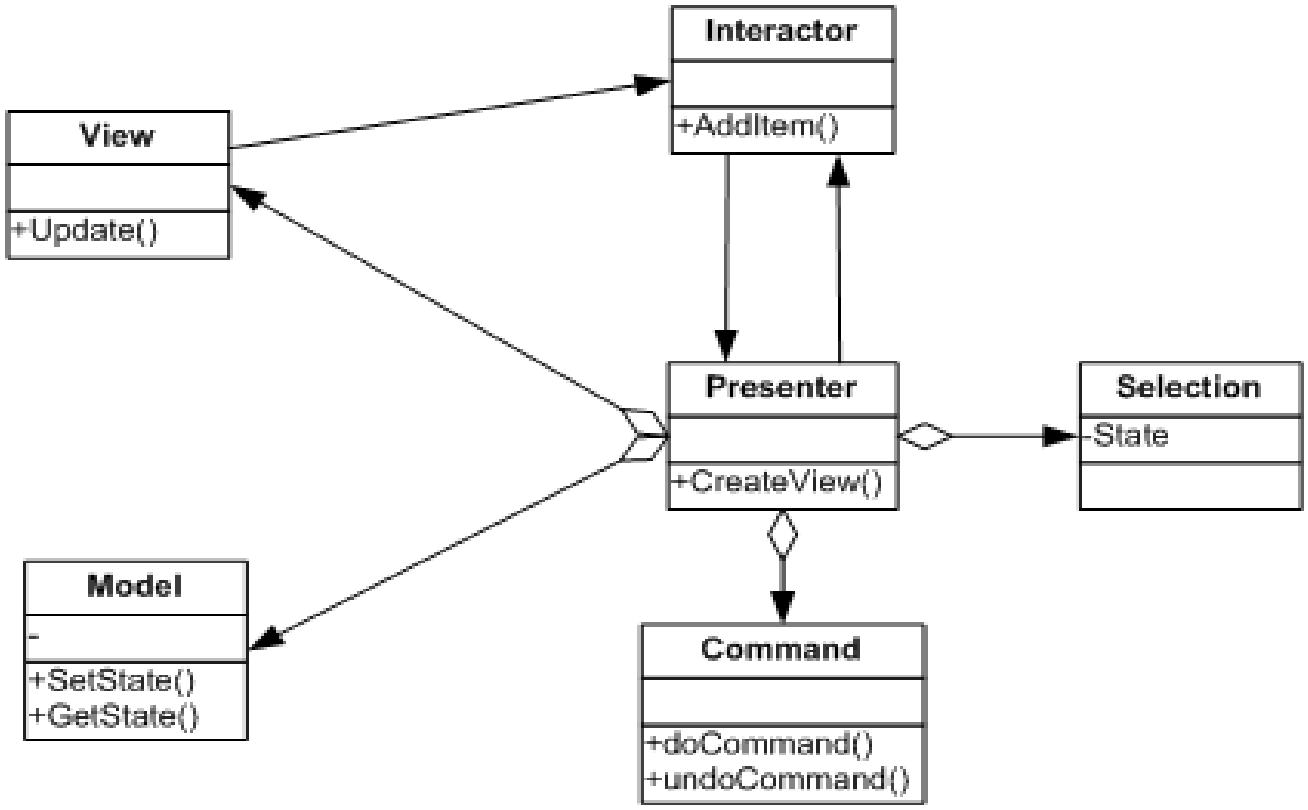
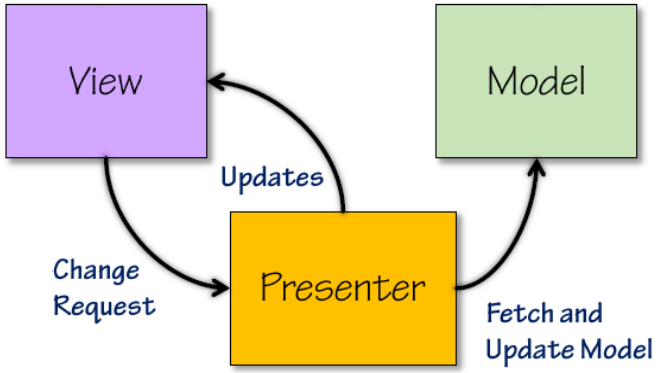


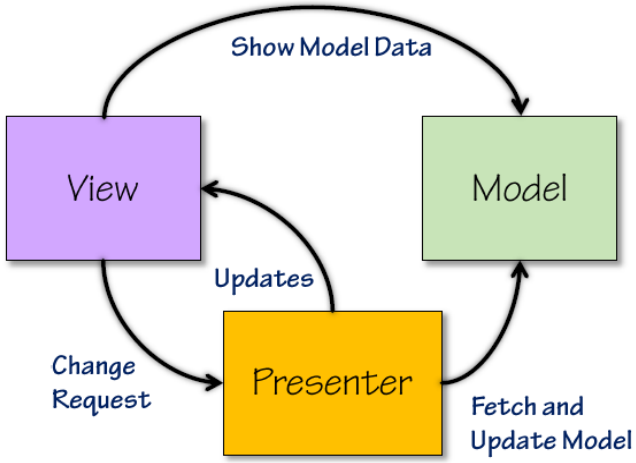
Figure 6: MVP

VARIANTS

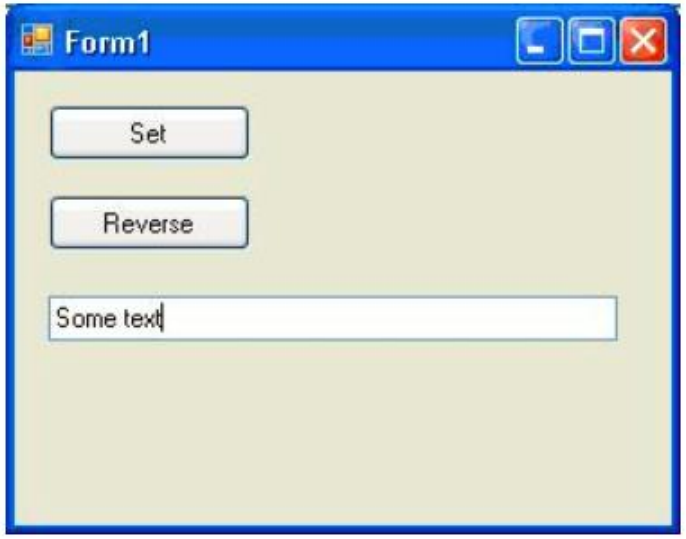
- View and Model are decoupled
- Presenter coordinates events between View and Model
- Passive View



- View and Model no longer separated
- Presenter coordinates events between View and Model
- Model contains only what View needs
- Presenter interacts with Domain Layer/Data Access



MVP EXAMPLE




```
namespace ModelViewPresenter
{
    public partial class Form1 : Form, IView
    {
        private Presenter presenter = null;
        private readonly Model m_Model;

        public Form1(Model model)
        {
            m_Model = model;
            InitializeComponent();
            presenter = new Presenter(this, m_Model);
            SubscribeToModelEvents();
        }

        public string TextValue
        {
            get
            {
                return textBox1.Text;
            }
            set
            {
                textBox1.Text = value;
            }

        }

        private void Set_Click(object sender, EventArgs e)
        {
            presenter.SetTextValue();
        }

        private void Reverse_Click(object sender, EventArgs e)
        {
            presenter.ReverseTextValue();
        }

        private void SubscribeToModelEvents()
        {
            m_Model.TextSet += m_Model_TextSet;
        }

        void m_Model_TextSet(object sender, CustomArgs e)
        {
            this.textBox1.Text = e.m_after;
            this.label1.Text = "Text changed from " + e.m_before + " to " + e.m_after;
        }
    }
```

Model injection in constructor

Creates Presenter

http://www.technical-recipes.com/2015/the-model-view-presenter-pattern-in-c-a-minimalist-implementation/

```
namespace ModelViewPresenter
{
    public class Presenter
    {
        private readonly IView m_View;
        private IModel m_Model;

        public Presenter(IView view, IModel model)
        {
            this.m_View = view;
            this.m_Model = model;
        }

        public void ReverseTextValue()
        {
            string reversed = ReverseString(m_View.TextValue);
            m_Model.Reverse(reversed);
        }

        public void SetTextValue()
        {
            m_Model.Set(m_View.TextValue);
        }

        private static string ReverseString(string s)
        {
            char[] arr = s.ToCharArray();
            Array.Reverse(arr);
            return new string(arr);
        }
    }
}
```

```
namespace ModelViewPresenter
{
    public class Model : IModel
    {
        private string m_textValue;

        public event EventHandler<CustomArgs> TextSet;
        public event EventHandler<CustomArgs> TextReverse;

        public Model()
        {
            m_textValue = "";
        }

        public void Set(string value)
        {
            string before = m_textValue;
            m_textValue = value;
            RaiseTextSetEvent(before, m_textValue);
        }

        public void Reverse(string value)
        {
            string before = m_textValue;
            m_textValue = value;
            RaiseTextSetEvent(before, m_textValue);
        }

        public void RaiseTextSetEvent(string before, string after)
        {
            TextSet(this, new CustomArgs(before, after));
        }
    }

    public class CustomArgs : EventArgs
    {
        public string m_before { get; set; }
        public string m_after { get; set; }

        public CustomArgs(string before, string after)
        {
            m_before = before;
            m_after = after;
        }
    }
}
```

MVVM

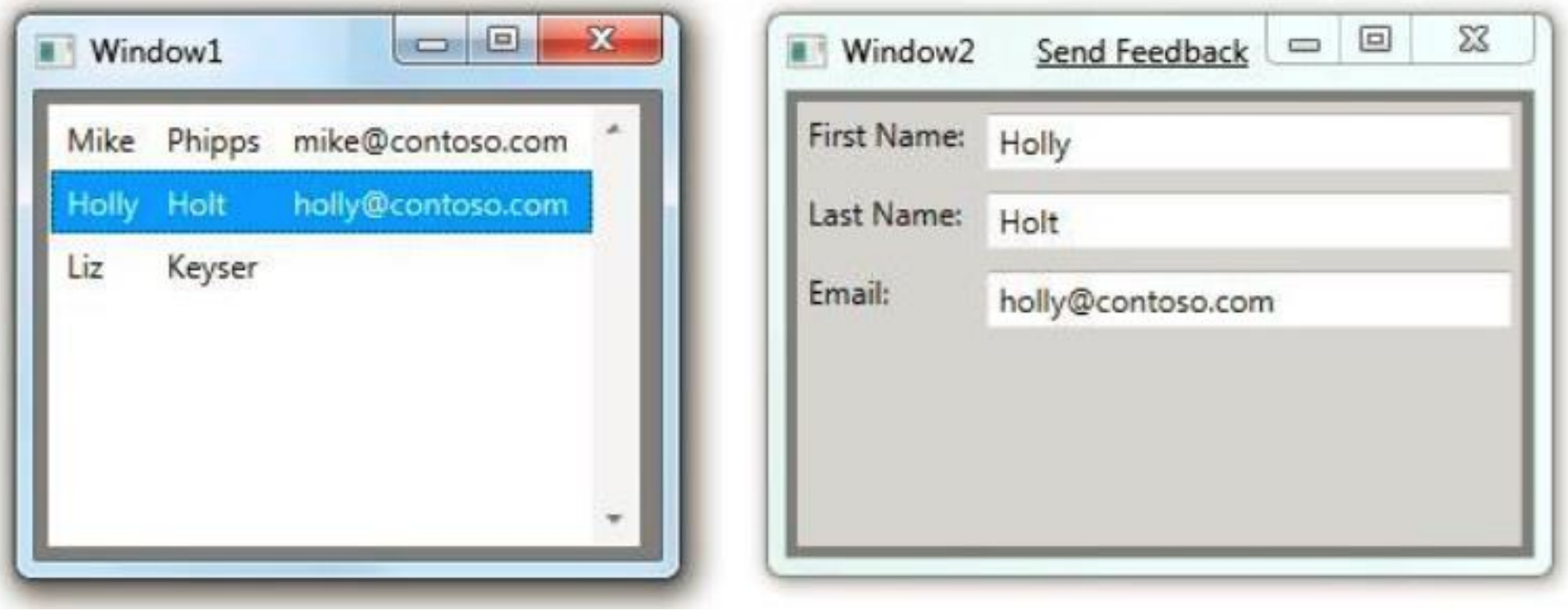
Motivation

- Need to share a project with a designer, and flexibility for design work and development work to happen near-simultaneously
- Thorough unit testing for your solutions
- Important to have reusable components, both within and across projects
- Flexibility to change the user interface without having to refactor other logic in the code base

https://www.wintellect.com/model-view-viewmodel-mvvm-explained/

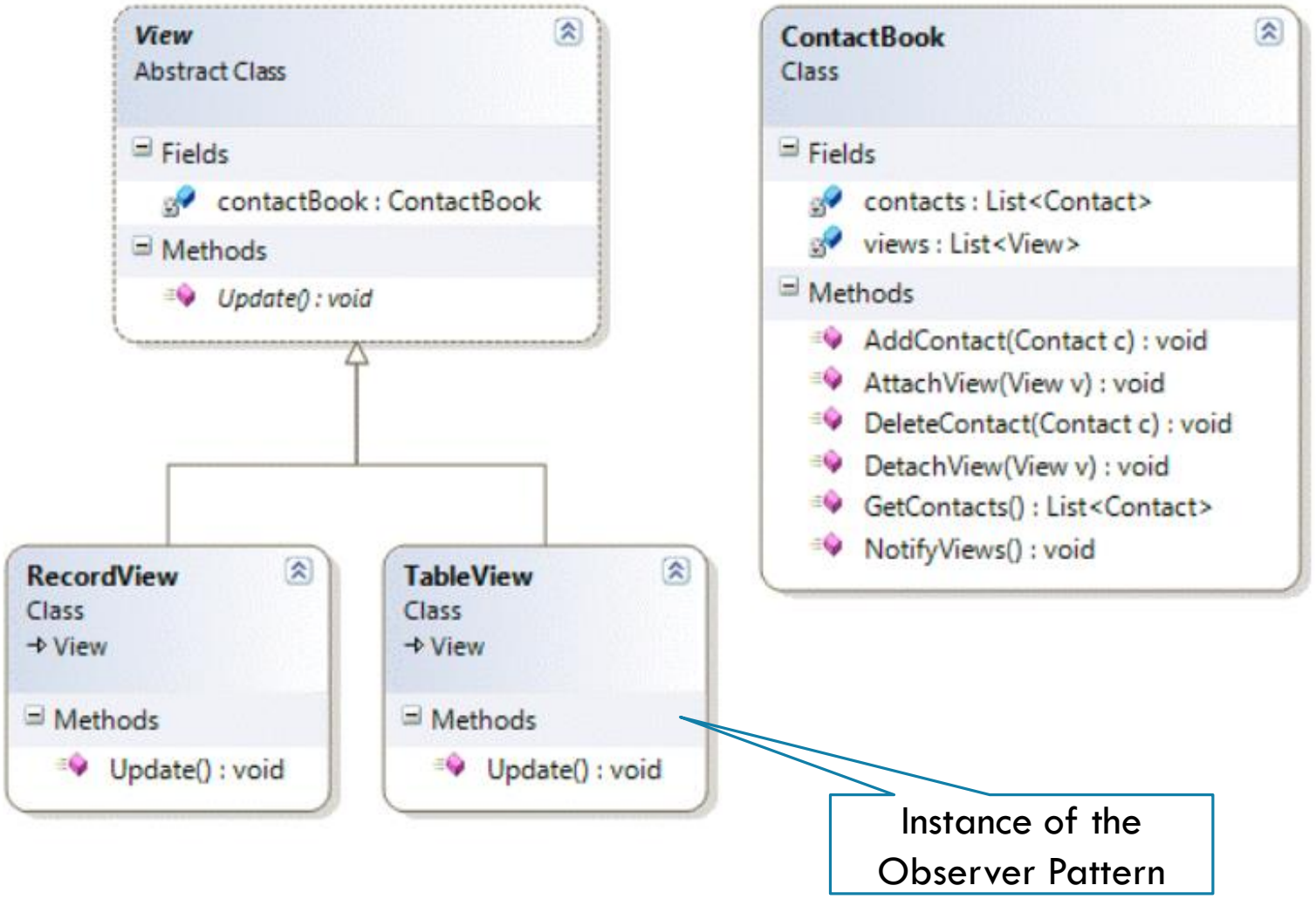
MV* EXAMPLE

UI Contacts App



[https://blogs.msdn.microsoft.com/ivo_manolov/2012/03/17/model-view-viewmodel-mvvm-applications-general-introduction/]

M-V SEPARATION



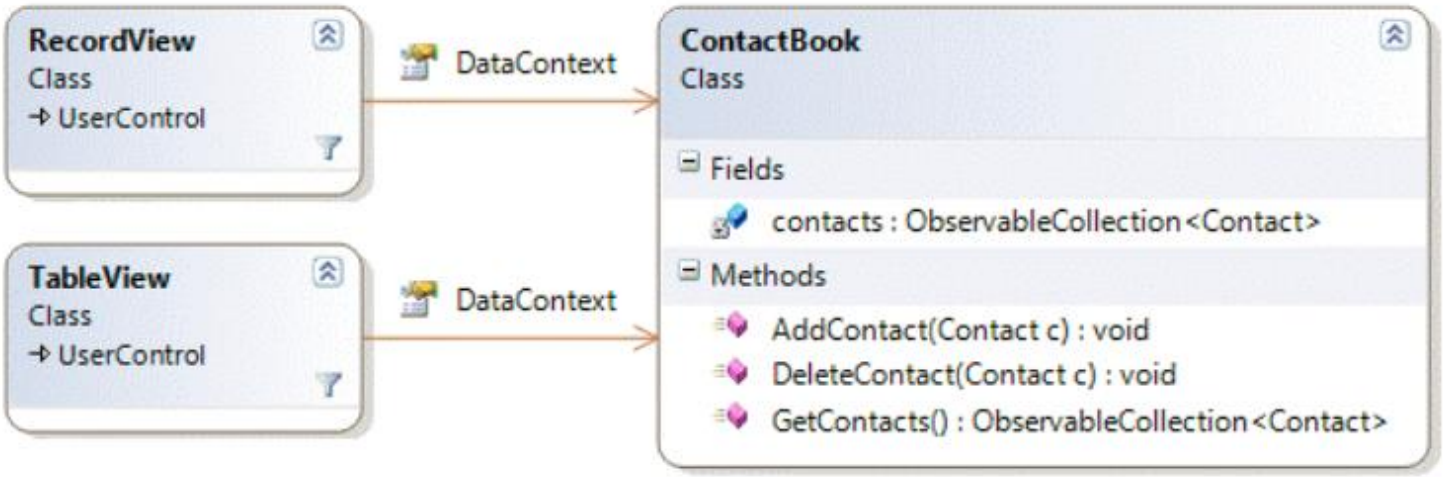
SIMPLISTIC WPF /SILVERLIGHT DESIGN

New features:

1. Track the selected item, so that we update RecordView whenever the selection in TableView changes, and vice versa.
2. Enable or disable parts of the UI of RecordView and TableView based on some rule (for example, highlight any entry that has an e-mail in the live.com domain).

Databinding is the ability to bind UI elements to any data.

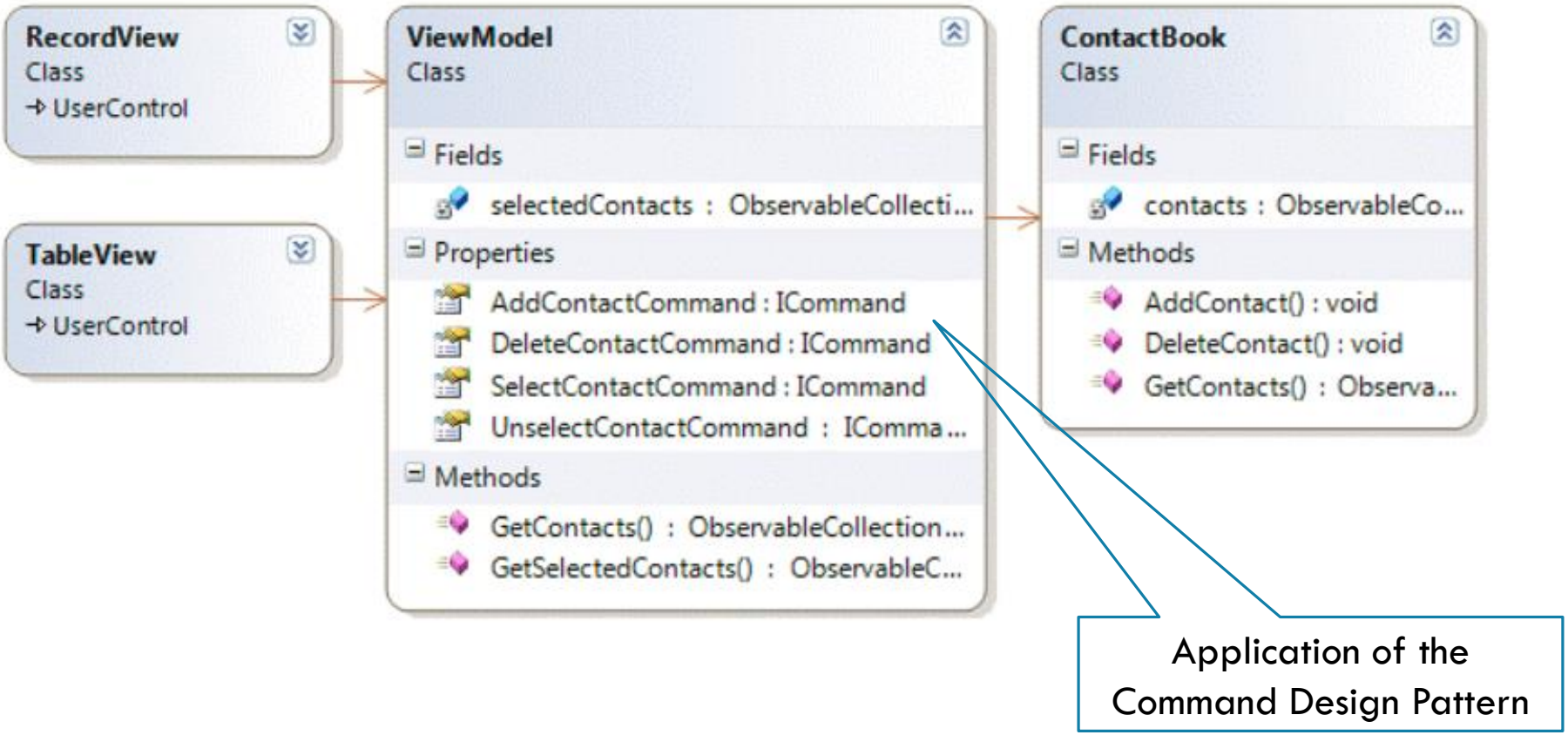
Commands provide the ability to notify the underlying data of changes in the UI.



MVVM DESIGN

- The views know of the ViewModel and bind to its data, to be able to reflect any changes in it.
- The ViewModel has no reference to the views—it holds only a reference to the model.
- For the views, the ViewModel acts both as
  - a façade to the model,
  - a way to share state between views (selectedContacts in the example).
- The ViewModel often exposes commands that the views can bind to and trigger.

COMMANDS IN MVVM APPS



MVVM

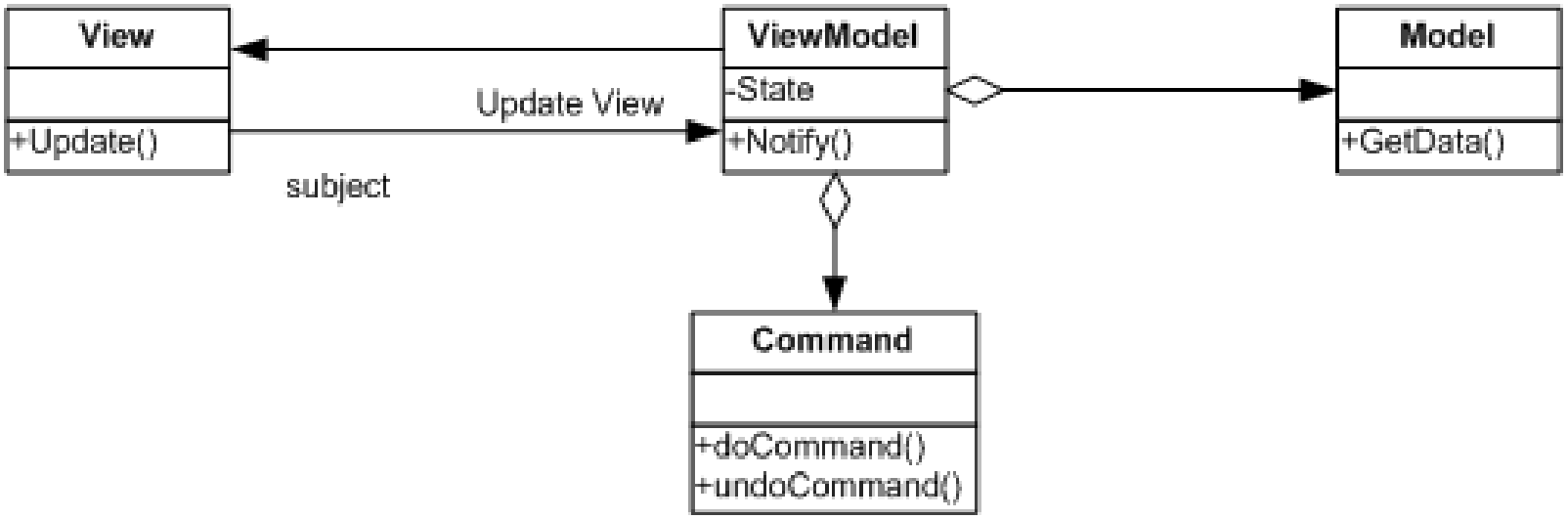
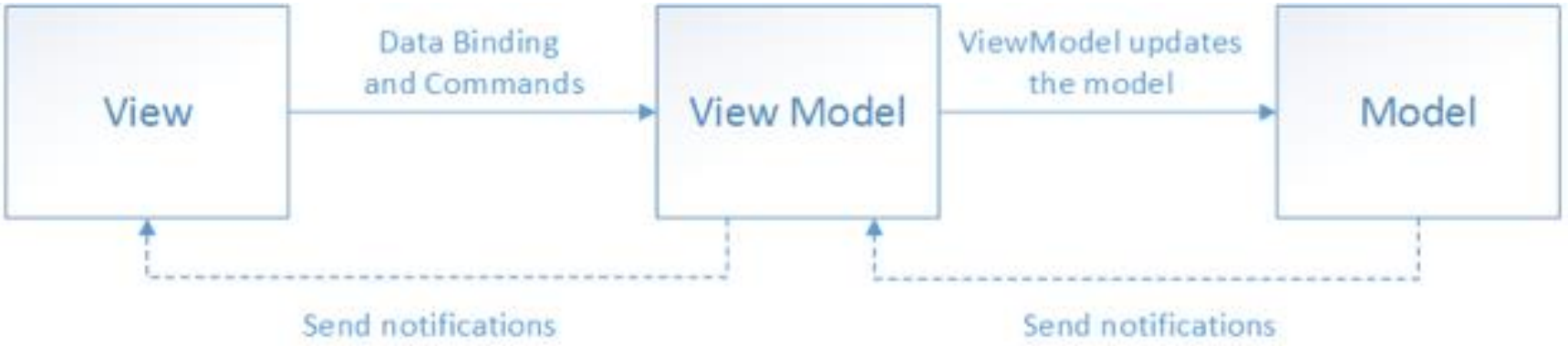


Figure 8: MVVM

https://www.wintellect.com/model-view-viewmodel-mvvm-explained/

MVVM STRUCTURE



Model

- Represents the data
  - 2 perspectives
    - OO approach: Domain Objects
    - Data-centric approach: XML, DAL, etc.
- No knowledge of how and where it will be presented
- Implements INotifyPropertyChanged

MVVM STRUCTURE

ViewModel

- Contains Properties, Commands, etc. to facilitate communication between View and Model
- Properties that expose instances of Model objects
- Commands and events that interact with Views for user interactions
- Implements INotifyPropertyChanged
- Pushes data up to the View

View

- Visual display
- Buttons, windows, graphics, etc.
- Responsible for displaying data
- Not responsible for retrieving data, business logic, business rules or validation
- Contains binding extensions
  - Identify data points represented to the user
  - Point to the names of the data point properties

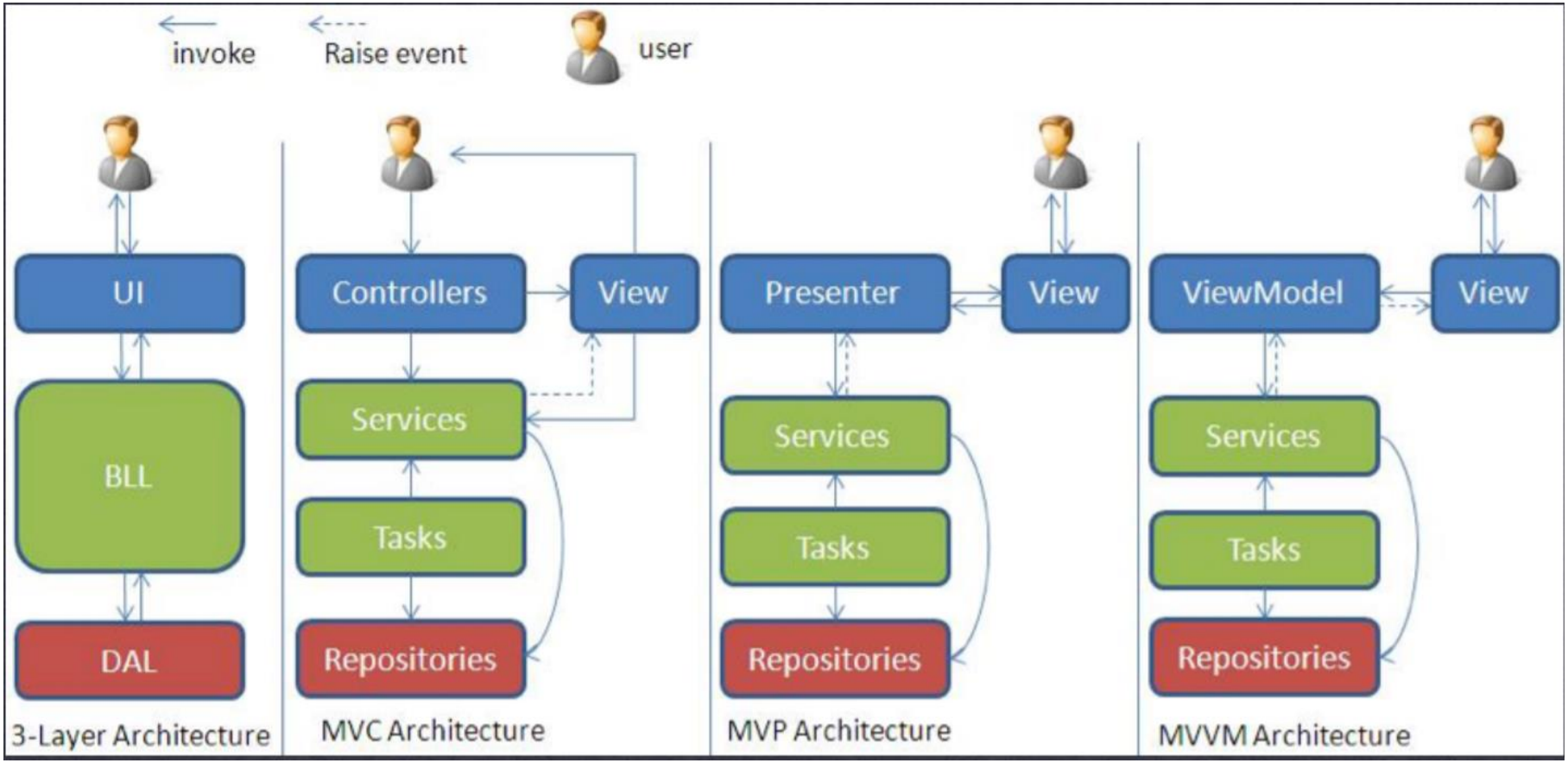
MVVM DISCUSSION

- MVVM is a derivative of MVC that takes advantage of particular strengths of the Windows Presentation Foundation (WPF) architecture
- Separates the Model and the View by introducing an abstract layer between them: a "View of the Model," or ViewModel.
- The typical relationship between a ViewModel and the corresponding Views is one-to-many, but not always.
- There are situations when one ViewModel is aware of another ViewModel within the same application. When this happens, one ViewModel can represent a collection of ViewModels.

MVVM BENEFITS

- A ViewModel provides a **single store for presentation** policy and state, thus improving the reusability of the Model (by decoupling it from the Views) and the replaceability of the Views (by removing specific presentation policy from them).
- MVVM improves the overall **testability** of the application.
- MVVM also improves the "**mockability**" of the application.
- MVVM is a very **loosely coupled** design. The View holds a reference to the ViewModel, and the ViewModel holds a reference to the Model. The rest is done by the data-binding and commanding infrastructure of WPF.

MV* IN A NUTSHELL



http://merroun.wordpress.com/2012/03/28/mvvm-mvp-and-mvc-software-patterns-againts-3-layered-architecture/

WRAP-UP

Architectural Patterns

- High-level design of an architecture
- Patterns may have several variants
- Need to be adapted/customized for the specific needs
- Use more basic design patterns