WEB APP ARCHITECTURES:

MULTI-TIER (2-TIER, 3-TIER)
MODEL-VIEWER-CONTROLLER (MVC)
REST ARCHITECTURAL STYLE

Slides created by Manos Papagelis

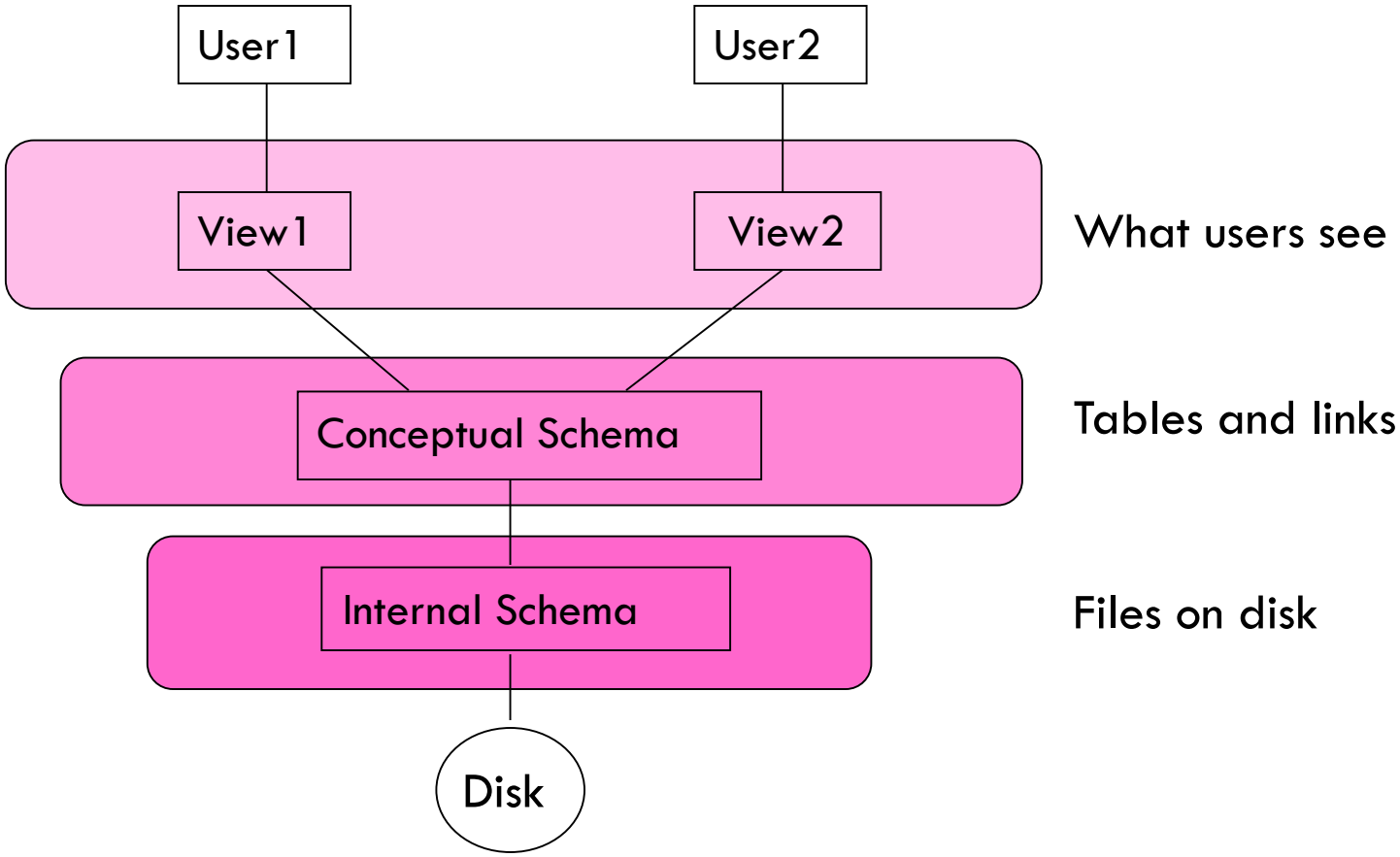
Based on materials by Marty Stepp, M. Ernst, S. Reges, D. Notkin, R. Mercer, R. Boswell,Wikipedia

Overview

- Data Independence in Relational Databases
- N-tier Architectures
- Design Patterns
  - The MVC Design Pattern
- REST Architectural Style

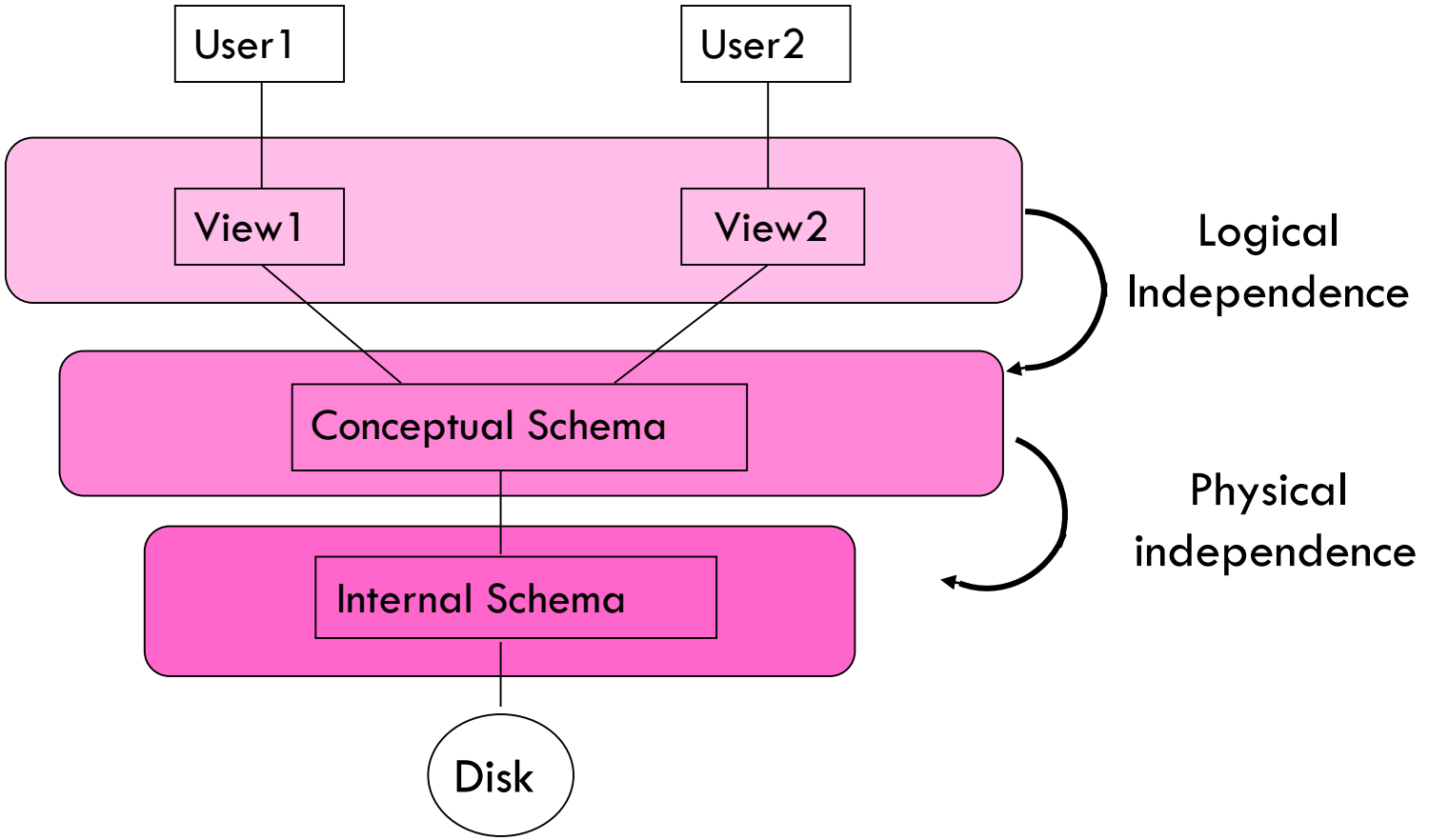
Data Independence in Rel. DBMS

Database Architecture With Views



Each level is independent of the levels below

Logical and Physical Independence



Each level is independent of the levels below

Data Independence

- **Logical Independence:** The ability to change the logical schema without changing the external schema or application programs
  - Can add new fields, new tables without changing views
  - Can change structure of tables without changing view

- **Physical Independence:** The ability to change the physical schema without changing the logical schema
  - Storage space can change
  - Type of some data can change for reasons of optimization

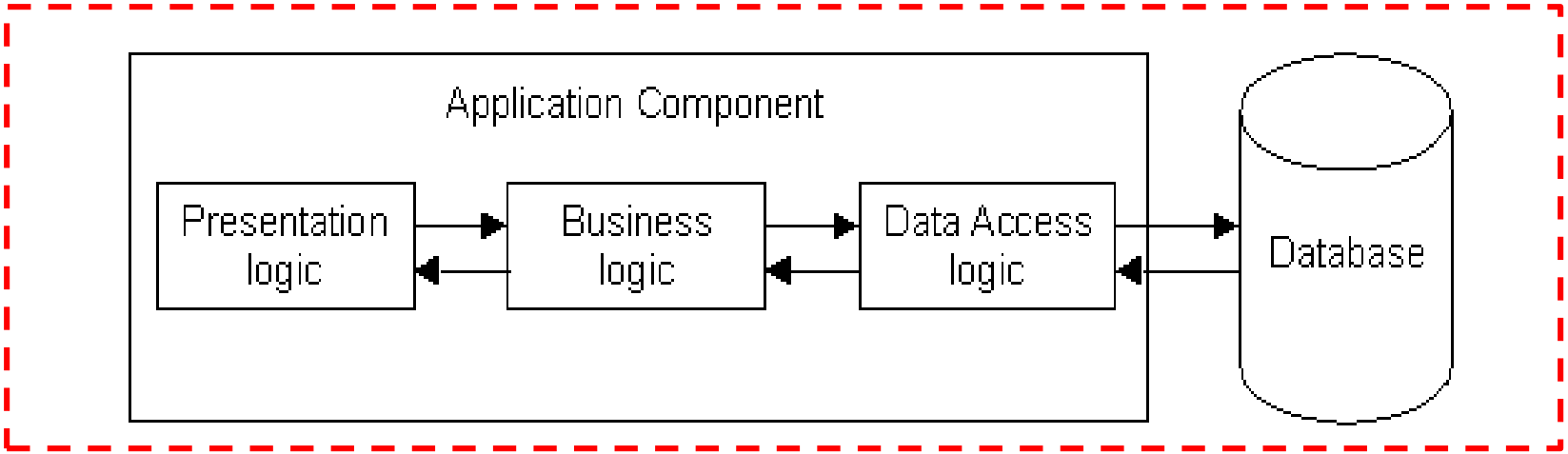
LESSON: Keep the VIEW (what the user sees) independent of the MODEL (domain knowledge)

N-tier architectures

Significance of “Tiers”

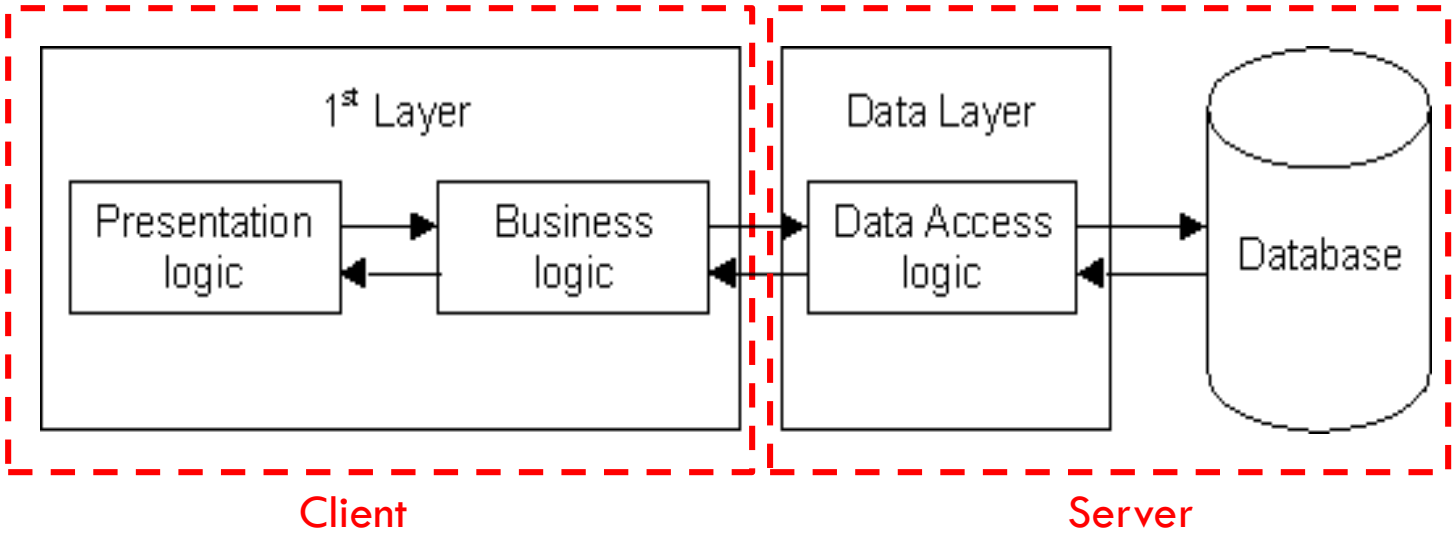
- N-tier architectures have the same components
  - Presentation
  - Business/Logic
  - Data
- N-tier architectures try to separate the components into different tiers/layers
  - Tier: physical separation
  - Layer: logical separation

1-Tier Architecture



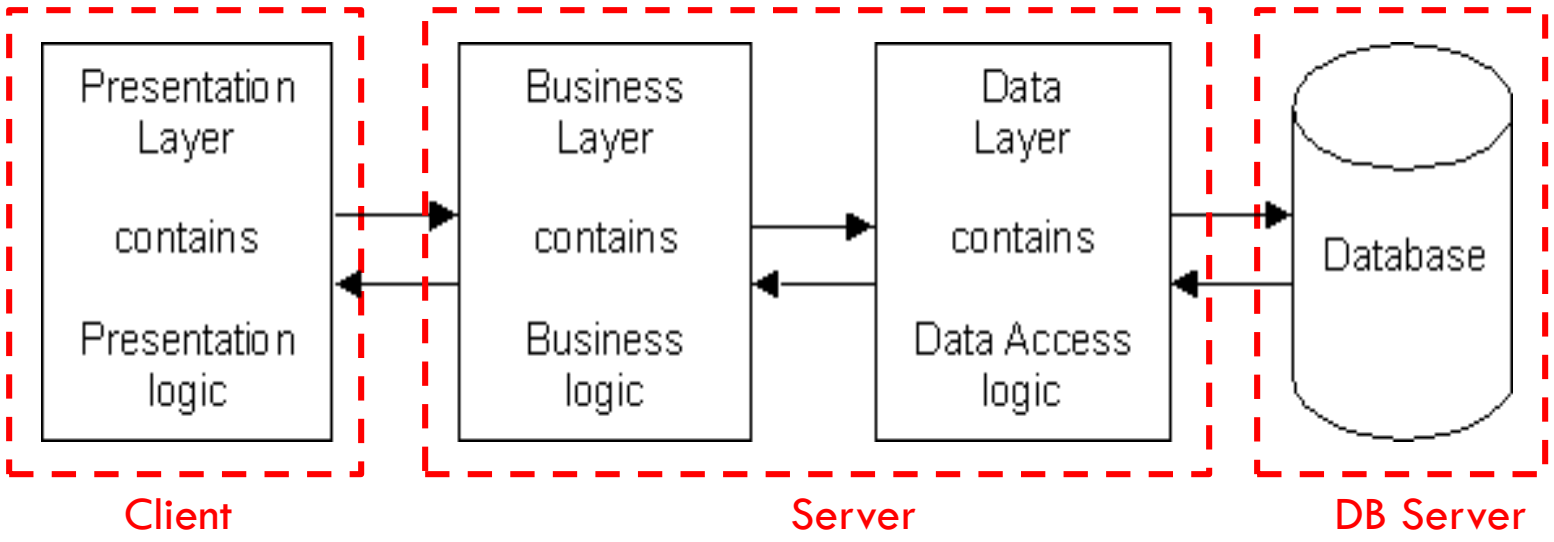
- All 3 layers are on the same machine
  - All code and processing kept on a single machine
- Presentation, Logic, Data layers are tightly connected
  - Scalability: Single processor means hard to increase volume of processing
  - Portability: Moving to a new machine may mean rewriting everything
  - Maintenance: Changing one layer requires changing other layers

2-Tier Architecture



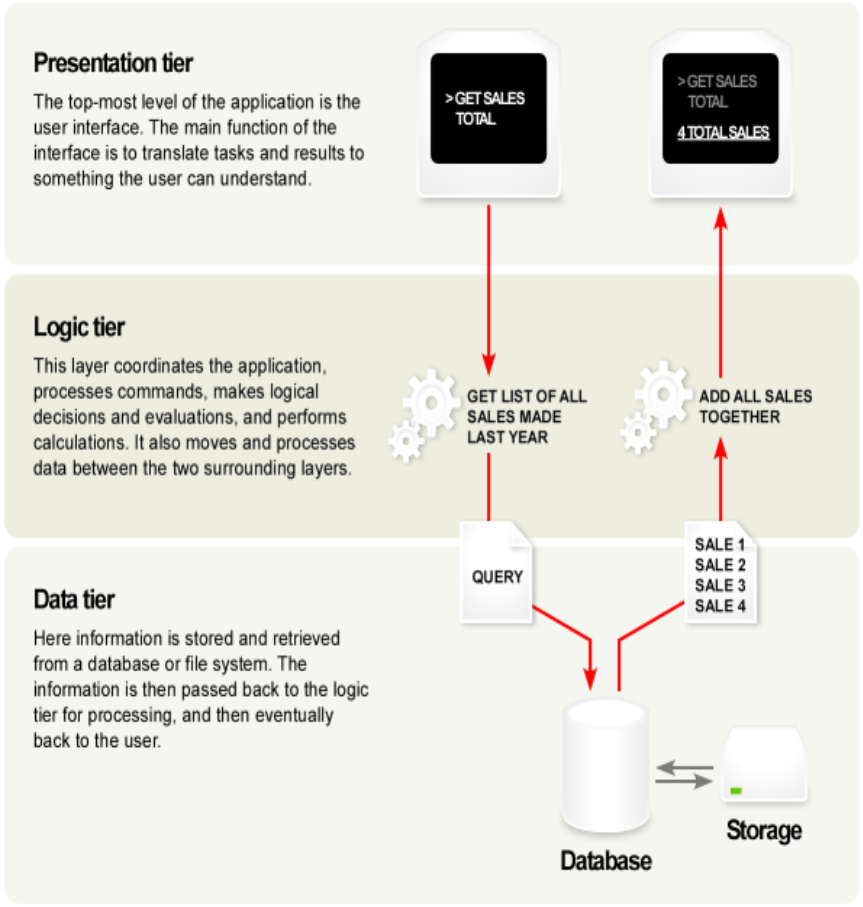
- Database runs on Server
  - Separated from client
  - Easy to switch to a different database
- Presentation and logic layers still tightly connected
  - Heavy load on server
  - Potential congestion on network
  - Presentation still tied to business logic

3-Tier Architecture



- Each layer can potentially run on a different machine
- Presentation, logic, data layers disconnected

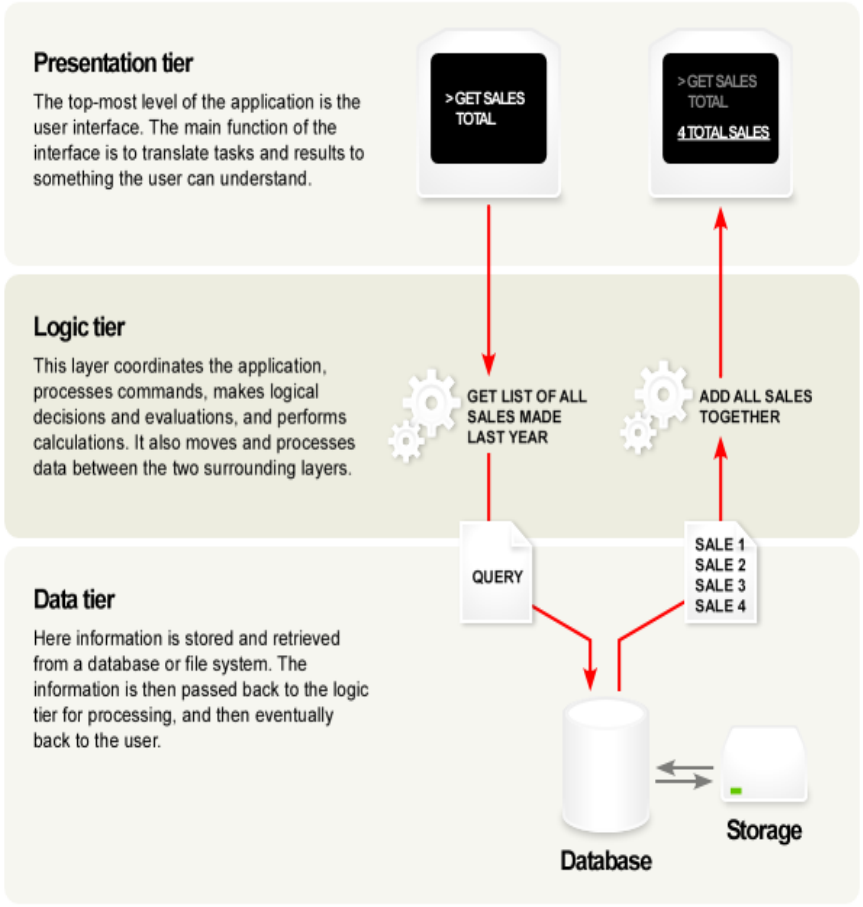
A Typical 3-tier Architecture



Architecture Principles

- Client-server architecture
- Each tier (Presentation, Logic, Data) should be independent and should not expose dependencies related to the implementation
- Unconnected tiers should not communicate
- Change in platform affects only the layer running on that particular platform

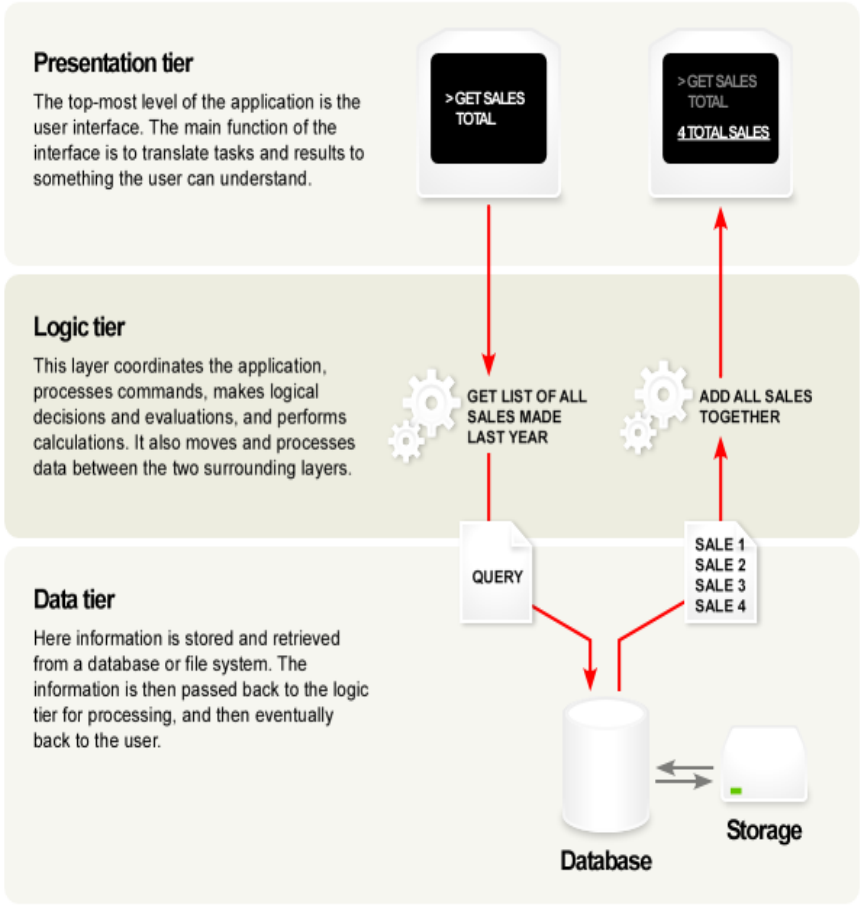
A Typical 3-tier Architecture



Presentation Layer

- Provides user interface
- Handles the interaction with the user
- Sometimes called the GUI or client view or front-end
- Should not contain business logic or data access code

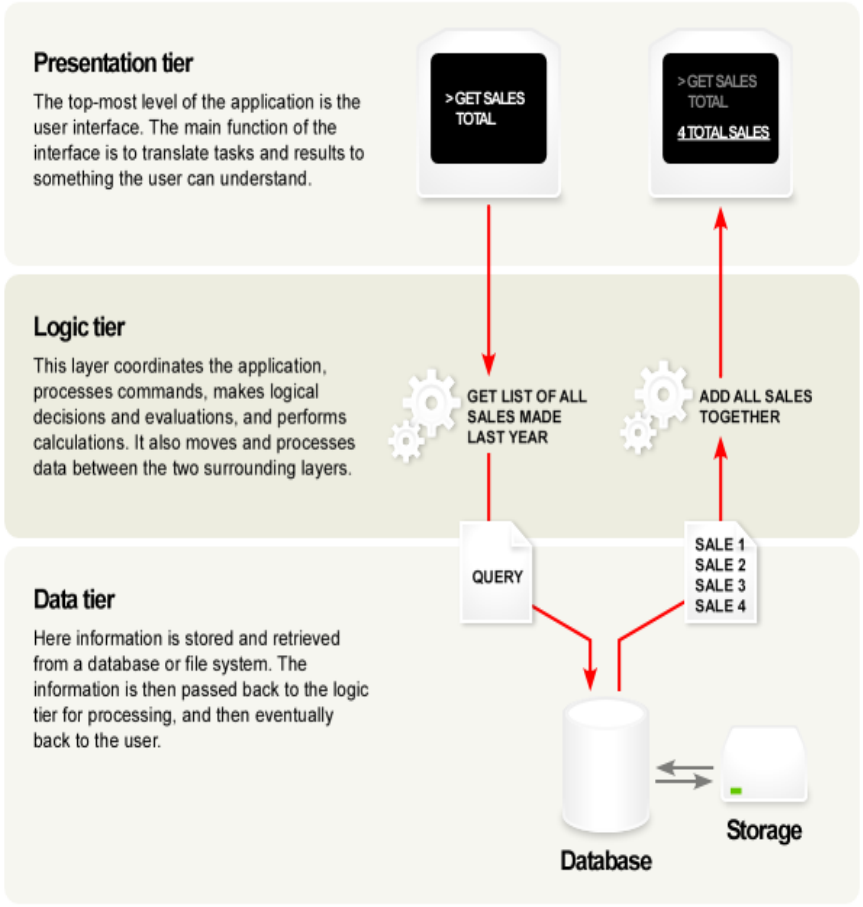
A Typical 3-tier Architecture



Logic Layer

- The set of rules for processing information
- Can accommodate many users
- Sometimes called middleware/back-end
- Should not contain presentation or data access code

A typical 3-tier Architecture



Data Layer

- The physical storage layer for data persistence
- Manages access to DB or file system
- Sometimes called back-end
- Should not contain presentation or business logic code

The 3-Tier Architecture for Web Apps

- Presentation Layer

  Static or dynamically generated content rendered by the browser (front-end)

- Logic Layer

  A dynamic content processing and generation level application server, e.g., Java EE, ASP.NET, PHP, ColdFusion platform (middleware)

- Data Layer

  A database, comprising both data sets and the database management system or RDBMS software that manages and provides access to the data (back-end)

3-Tier Architecture - Advantages

- Independence of Layers
  - Easier to maintain
  - Components are reusable
  - Faster development (division of work)
    - Web designer does presentation
    - Software engineer does logic
    - DB admin does data model

Design Patterns

Design Problems & Decisions

- Construction and testing
  - how do we build a web application?
  - what technology should we choose?
- Re-use
  - can we use standard components?
- Scalability
  - how will our web application cope with large numbers of requests?
- Security
  - how do we protect against attack, viruses, malicious data access, denial of service?
- Different data views
  - user types, individual accounts, data protection

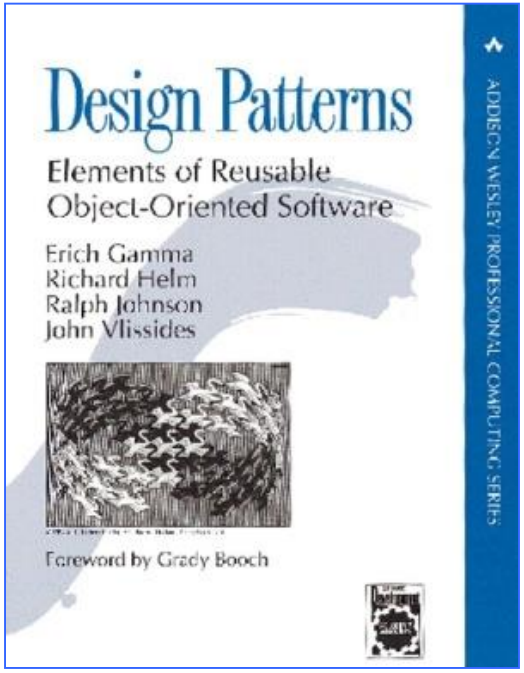
Need for general and reusable solution: **Design Patterns**

What is a Design Pattern?

- A general and reusable solution to a commonly occurring problem in the design of software
- A template for how to solve a problem that has been used in many different situations
- NOT a finished design
  - the pattern must be adapted to the application
  - cannot simply translate into code

Origin of Design Patterns

- Architectural concept by Christopher Alexander (1977/79)
- Adapted to OO Programming by Beck and Cunningham (1987)
- Popularity in CS after the book: “Design Patterns: Elements of Re-useable Object-oriented software”, 1994. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Now widely-used in software engineering



The MVC Design Pattern

Design Problem

- Need to change the look-and-feel without changing the core/logic
- Need to present data under different contexts (e.g., powerful desktop, web, mobile device).
- Need to interact with/access data under different contexts (e.g., touch screen on a mobile device, keyboard on a computer)
- Need to maintain multiple views of the same data (list, thumbnails, detailed, etc.)

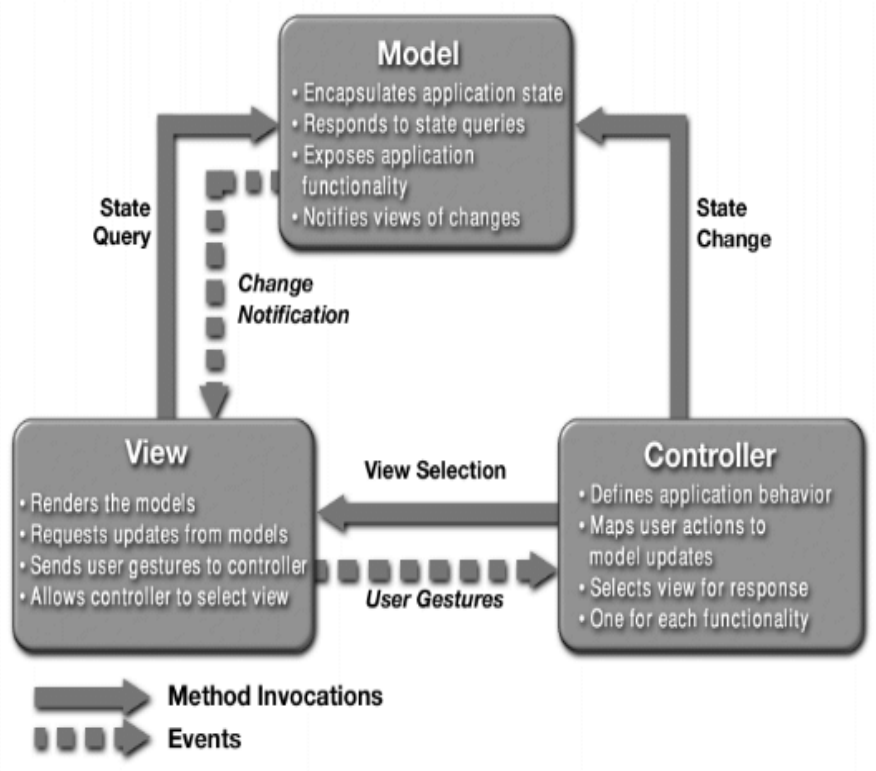
Design Solution

- Separate core functionality from the presentation and control logic that uses this functionality
- Allow multiple views to share the same data model
- Make supporting multiple clients easier to implement, test, and maintain

The Model-View-Controller Pattern

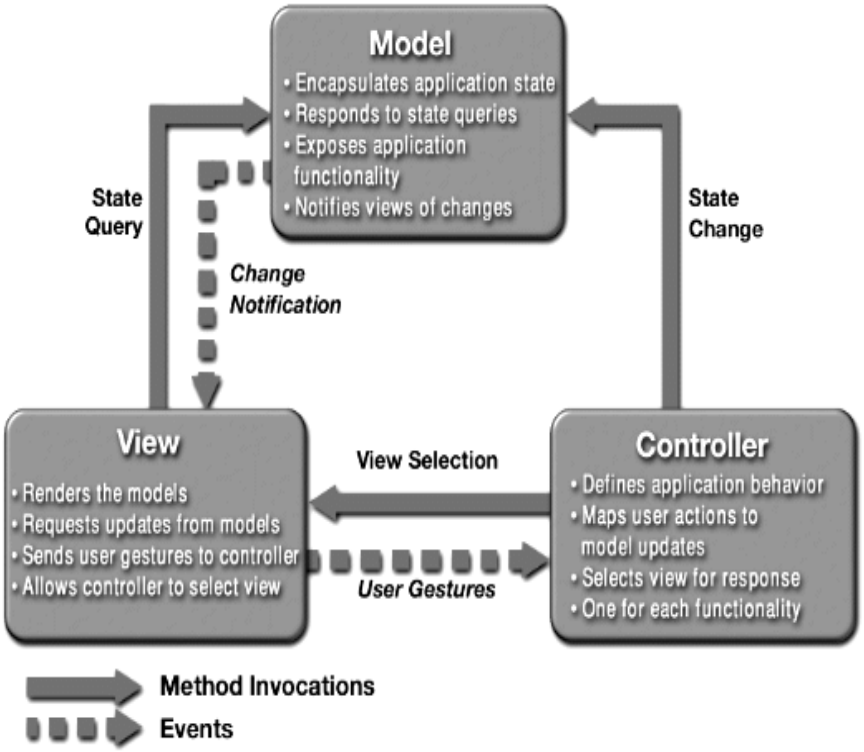
Design pattern for graphical systems that promotes separation between model and view

With this pattern the logic required for data **maintenance** (database, text file) is separated from how the data is **viewed** (graph, numerical) and how the data can be **interacted with** (GUI, command line, touch)



The MVC Pattern

- Model
  - manages the behavior and data of the application domain
  - responds to requests for information about its state (usually from the view)
  - follows instructions to change state (usually from the controller)
- View
  - renders the model into a form suitable for interaction, typically a user interface (multiple views can exist for a single model for different purposes)
- Controller
  - receives user input and initiates a response by making calls on model objects
  - accepts input from the user and instructs the model and viewport to perform actions based on that input



The MVC Pattern (in practice)

- Model
  - Contains domain-specific knowledge
  - Records the state of the application
    - E.g., what items are in a shopping cart
  - Often linked to a database
  - Independent of view
    - One model can link to different views
- View
  - Presents data to the user
  - Allows user interaction
  - Does no processing
- Controller
  - defines how user interface reacts to user input (events)
  - receives messages from view (where events come from)
  - sends messages to model (tells what data to display)

The MVC for Web Applications

- Model
  - database tables (persistent data)
  - session information (current system state data)
  - rules governing transactions
- View
  - (X)HTML
  - CSS style sheets
  - server-side templates
- Controller
  - client-side scripting
  - http request processing
  - business logic/preprocessing

MVC Advantages

- Clarity of Design
  - model methods give an API for data and state
  - eases the design of view and controller
- Efficient Modularity
  - any of the components can be easily replaced
- Multiple Views
  - many views can be developed as appropriate
  - each uses the same API for the model
- Easier to Construct and Maintain
  - simple (text-based) views while constructing
  - more views and controllers can be added
  - stable interfaces ease development
- Distributable
  - natural fit with a distributed environment

3-tier Architecture vs. MVC Architecture

- Communication
  - **3-tier:** The presentation layer never communicates directly with the data layer-only through the logic layer (linear topology)
  - **MVC:** All layers communicate directly (triangle topology)
- Usage
  - **3-tier:** Mainly used in web applications where the client, middleware and data tiers ran on physically separate platforms
  - **MVC:** Historically used on applications that run on a single graphical workstation (applied to separate platforms as *Model 2*)

REST ARCHITECTURAL STYLE

REST API & RESTful Web Services

Slides by Ahmed Shah Mashiyat

REST - Representational State Transfer

"REST is just a set of conventions about how to use HTTP"

Instead of having randomly named setter and getter URLs and using GET for all the getters and POST for all the setters, we try to have the URLs identify resources, and then use the HTTP actions GET, POST, PUT and DELETE to do stuff to them. So instead of

```
GET /get_article?id=1
```

```
POST /delete_article?id=1
```

You would do

```
GET /articles/1/
```

```
DELETE /articles/1/
```

http://stackoverflow.com/questions/2191049/what-is-the-advantage-of-using-rest-instead-of-non-rest-http

REST API

We need two basic URLs per resource

→ One for a collection of the resource

```
/dogs
```

→ The other one is for a particular resource.

```
/dogs/1
```

REST API

GET

Read a specific resource (by an identifier) or a collection of resources.

PUT

Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is know before hand.

REST API

DELETE

Remove/delete a specific resource by an identifier.

POST

Create a new resource. Also a catch-all verb for operations that don't fit into the other categories.

REST API

To show association (collection within a collection):

```
/owner/123/dogs
```

Or

A bit more complex

```
/dogs?color=white&location=toronto
```

REST API - Idempotence

Same result over unlimited number of calls, since we are dealing with same resources.

However, *delete* may return 404 error in subsequent calls if not handled properly.

REST API – Best Practices

→ think about “resources”

→ elements & collections

→ map out the 4 methods for each

→ Prefer Nouns, Plurals, Concrete

→ Use Parameters for more advanced queries