SYSTEMS DESIGN

BASIC CONCEPTS

Systems design is the process of defining the architecture, modules, interfaces, and data for a system to satisfy specified requirements. Systems design could be seen as the application of systems theory to product development.

ARCHITECTURAL DESIGN

The architectural design of a system emphasizes the design of the system architecture that describes the structure, behavior and more views of that system and analysis.

LOGICAL DESIGN

The logical design of a system pertains to an abstract representation of the data flows, inputs and outputs of the system. This is often conducted via modelling, using an over-abstract (and sometimes graphical) model of the actual system. In the context of systems, designs are included. Logical design includes entity-relationship diagrams (ER diagrams).

PHYSICAL DESIGN

The physical design relates to the actual input and output processes of the system. This is explained in terms of how data is input into a system, how it is verified/authenticated, how it is processed, and how it is displayed.

The physical portion of system design can generally be broken down into three sub-tasks:

- User Interface Design
- Data Design
- Process Design

User Interface Design is concerned with how users add information to the system and with how the system presents information back to them. Data Design is concerned with how the data is represented and stored within the system. Finally, Process Design is concerned with how data moves through the system, and with how and where it is validated, secured and/or transformed as it flows into, through and out of the system. At the end of the system design phase, documentation describing the three sub-tasks is produced and made available for use in the next phase.

Physical design, in this context, does not refer to the tangible physical design of an information system. To use an analogy, a personal computer's physical design involves input via a keyboard, processing within the CPU, and output via a monitor, printer, etc. It would not concern the actual layout of the tangible hardware, which for a PC would be a monitor, CPU, motherboard, hard drive, modems, video/graphics cards, USB slots, etc.

MULTITIER ARCHITECTURE

In software engineering, multitier architecture (often referred to as n-tier architecture) or multilayered architecture is a client–server architecture in which presentation, application processing, and data management functions are physically separated. The most widespread use of multitier architecture is the three-tier architecture.

N-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application. A three-tier architecture is typically composed of a presentation tier, a domain logic tier, and a data storage tier.

While the concepts of layer and tier are often used interchangeably, one fairly common point of view is that there is indeed a difference. This view holds that a layer is a logical structuring mechanism for the elements that make up the software solution, while a tier is a physical structuring mechanism for the system infrastructure. For example, a three-layer solution could easily be deployed on a single tier, such as a personal workstation.

THREE-TIER ARCHITECTURE

Three-tier architecture is a client–server software architecture pattern in which the user interface (presentation), functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.

Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology. For example, a change of operating system in the presentation tier would only affect the user interface code.

Typically, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface, functional process logic that may consist of one or more separate modules running on a workstation or application server, and an RDBMS on a database server or mainframe that contains the computer data storage logic. The middle tier may be multitiered itself (in which case the overall architecture is called an "n-tier architecture").

Presentation tier

This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing and shopping cart contents. It communicates with other tiers by which it puts out the results to the browser/client tier and all other tiers in the network. In simple terms, it is a layer which users can access directly (such as a web page, or an operating system's GUI).

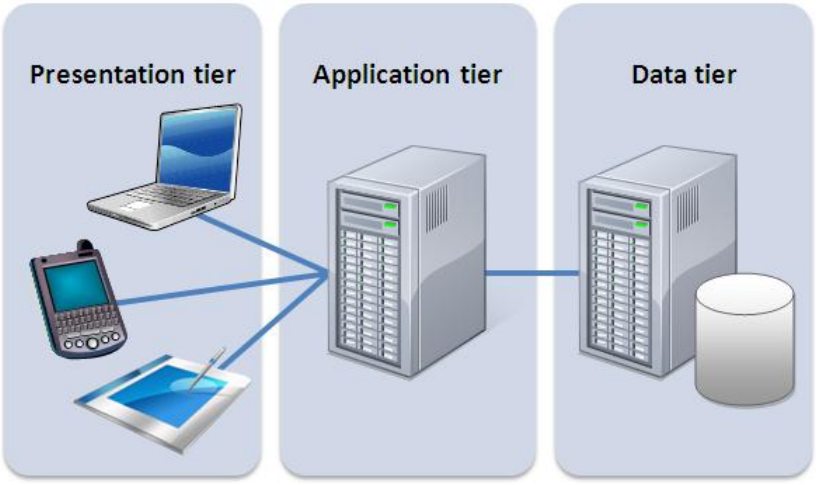
Application tier (business logic, logic tier, or middle tier)

The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application’s functionality by performing detailed processing.

Data tier

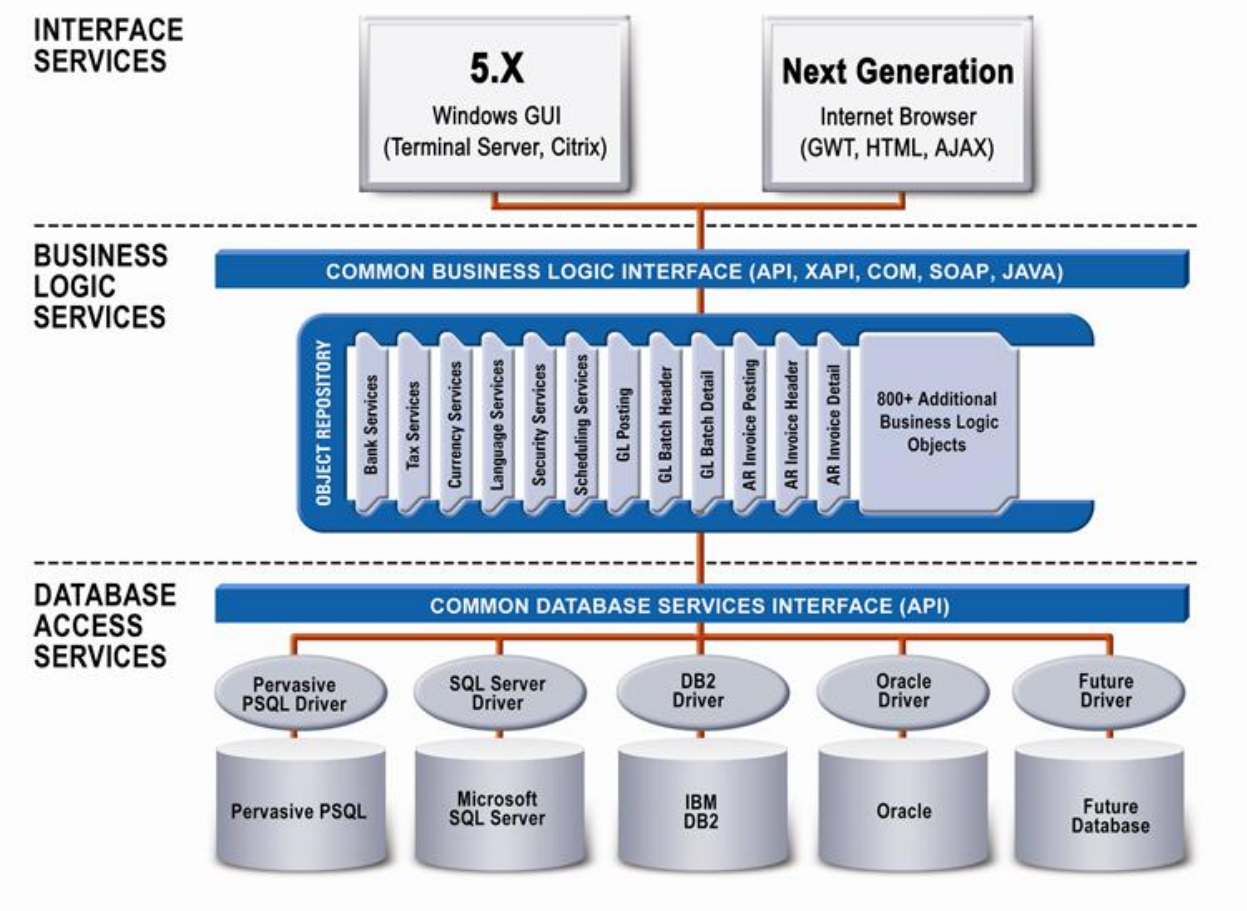
The data tier includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. The data access layer should provide an

API to the application tier that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms. Avoiding dependencies on the storage mechanisms allows for updates or changes without the application tier clients being affected by or even aware of the change. As with the separation of any tier, there are costs for implementation and often costs to performance in exchange for improved scalability and maintainability.



EXAMPLE DIAGRAMS

ACCOUNTING SYSTEM



Ref: https://smist08.wordpress.com/2010/09/11/accpacs-business-logic/

ERP SYSTEM

Ref: http://www.fatahss.com/fatah_erp_software.html

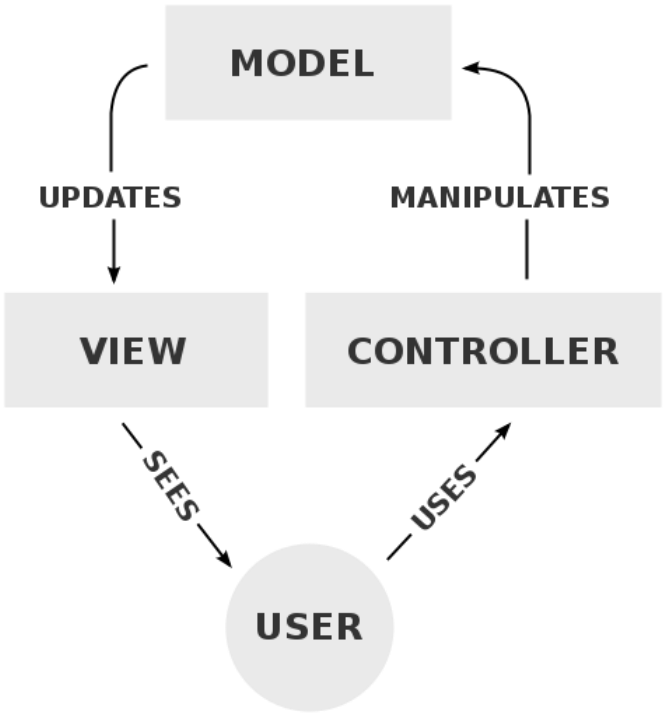
MVC DESIGN PATTERN

INTRODUCTION

Model–view–controller (MVC) is commonly used for developing software that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

Traditionally used for desktop graphical user interfaces (GUIs), this architecture has become popular for designing web applications and even mobile, desktop and other clients. Popular programming languages like Java, C#, Ruby, PHP and others have popular MVC frameworks that are currently being used in web application development straight out of the box.

The tools offered by Android, with layouts, Activities, and data structures, seem to steer us in the direction of the Model View Controller (MVC) pattern. MVC is a solid, established pattern that aims to isolate the different roles of an application. This is known as separation of concerns.



APPLICATIONS

The MVC pattern became popular with Java developers when WebObjects was ported to Java. Later frameworks for Java, such as Spring (released in October 2002), continued the strong bond between Java and MVC. The introduction of the frameworks Django (July 2005, for Python) and Rails (December 2005, for Ruby), both of which had a strong emphasis on rapid deployment, increased MVC's popularity outside the traditional enterprise environment in which it has long been popular. MVC web frameworks now hold large market-shares relative to non-MVC web toolkits.

USE IN WEB APPLICATIONS

Although originally developed for desktop computing, MVC has been widely adopted as an architecture for World Wide Web applications in major programming languages. Several web frameworks have been created that enforce the pattern. These software frameworks vary in their interpretations, mainly in the way that the MVC responsibilities are divided between the client and server.

Some web MVC frameworks take a thin client approach that places almost the entire model, view and controller logic on the server. This is reflected in frameworks such as Django, Rails and ASP.NET MVC. In this approach, the client sends either hyperlink requests or form submissions to the controller and then receives a complete and updated web page (or other document) from the view; the model exists entirely on the server. Other frameworks such as AngularJS, EmberJS, JavaScriptMVC and Backbone allow the MVC components to execute partly on the client (also see Ajax).

GOALS OF MVC

This section possibly contains original research. Please improve it by verifying the claims made and adding inline citations. Statements consisting only of original research should be removed. (February 2017) (Learn how and when to remove this template message)

Simultaneous development: Because MVC decouples the various components of an application, developers are able to work in parallel on different components without impacting or blocking one another. For example, a team might divide their developers between the front-end and the back-end. The back-end developers can design the structure of the data and how the user interacts with it without requiring the user interface to be completed. Conversely, the front-end developers are able to design and test the layout of the application prior to the data structure being available.

Code reuse: By creating components that are independent of each other, developers are able to reuse components quickly and easily in other applications. The same (or similar) view for one application can be refactored for another application with different data because the view is simply handling how the data is being displayed to the user.

ADVANTAGES

- Simultaneous development – Multiple developers can work simultaneously on the model, controller and views.
- High cohesion – MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Low coupling – The very nature of the MVC framework is such that there is low coupling among models, views or controllers
- Ease of modification – Because of the separation of responsibilities, future development or modification is easier
- Multiple views for a model – Models can have multiple views

DISADVANTAGES

- Code navigability – The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Multi-artifact consistency – Decomposing a feature into three artifacts causes scattering. Thus, requiring developers to maintain the consistency of multiple representations at once.
- Pronounced learning curve – Knowledge on multiple technologies becomes the norm. Developers using MVC need to be skilled in multiple technologies.