Webology (ISSN: 1735-188X)
Volume 18, Number 6, 2021

Analysis Of Various Software Architecture Process With Its Importance And Challenges While Developing In Real Scenario

[1]Gurudev Sawarkar , [2]Dr. Dipesh Rajput , [3]Dr. Tryambak Hiwarkar

[1]PhD Scholar, [2]Associate Professor, [3]Professor & Head.

[1,2,3] Department of Computer Science and Engineering.

[12,]Swami Vivekananda University Sagar MP.

[2] Wainganga College of Engineering Nagpur.

Abstract:

Among the most crucial responsibilities of an information system manager are the enhancement of programming productivity and the management of software expenditures. We say that a programmed has a structure if and only if its components are structured in a particular way.

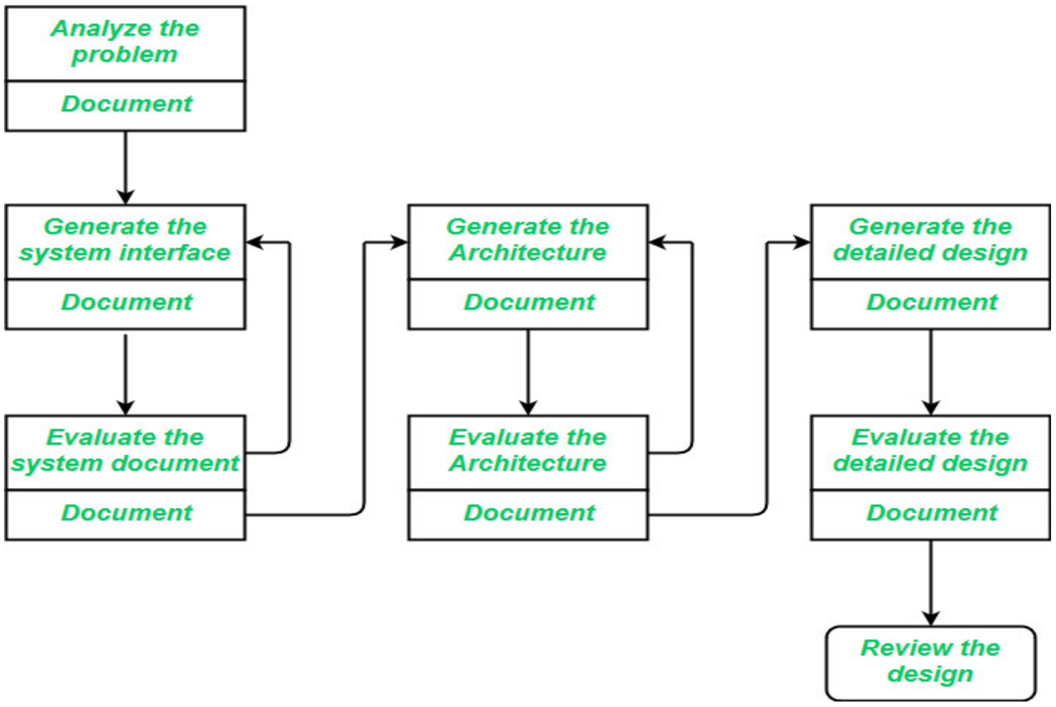
Software engineering (SE) practitioners consider software output to be a major area of study because it bridges the gap between technical and socioeconomic considerations. However, software development processes are intricate, and the term "productivity" has varying meanings depending on the context. That's why we need to in order to maximize the impact of software productivity studies on SE industry practice, the Existing practitioner perspectives and concerns need to be analyzed and synthesized. In this work, a mapping research is constructed to probe the potential for varying empirical impressions ofthe rate of production throughout the several fields of business and academic study that make up the industrial practice of SE, and to highlight their shared features.

Keywords: Interface Design, Architectural Design.

I. Software Design Process

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels of phases of design:

1. Interface Design
2. Architectural Design
3. Detailed Design


Analyze the problem
Document
Generate the system interface
Document
Evaluate the system document
Document
Generate the Architecture
Document
Evaluate the Architecture
Document
Generate the detailed design
Document
Evaluate the detailed design
Document
Review the design

II. Interface Design:

Interface design is the specification of the interaction between a system and its environment. this phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification on the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

III. Architectural Design:

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.

Issues in architectural design includes:

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between components.

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

Detailed Design:

Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

IV. Architectural Design of Software

The architectural design of software is necessary to represent its design. When it comes to computers, architectural design is defined by IEEE as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." Computing-based system software may display any of a wide variety of architectural motifs.

Every presentation mode will detail a distinct class of computer systems, which will include:

A collection of parts (such a database or a group of computing modules) that work together to accomplish a certain task for the system.

Coordinating, communicating, and working together amongst parts will be easier with this set of connections. A set of constraints dictating how individual parts of the system must fit together.

Models in the designer's domain of meaning that illuminate the system's general characteristics.

Architectural styles are used to provide a framework within which the many parts of a system can function.

4.1 Classification of Architectural elegances:

Facts placed designs:

Data centric designs have a data store at their core, which is regularly accessed by other parts of the system to make changes to the stored information.

The diagram depicts a common approach to presenting data. A centralized data store is accessed by the client programmed. When client-related data or client-interest data changes, the repository may be transformed into a blackboard using a variant of this technique.

The integration of systems will be facilitated by this data-centric design. What this means is that the current components can be modified and new client components added to the architecture without the need for consent from or worry over other clients. With the blackboard technique, information may be shared between users.

Data flow architectures:

Such an architecture is employed if the goal is to convert one set of input data into another set of output data via a chain of computational manipulating components.

The diagram depicts a pipe-and-filter architecture, so named because it employs both pipes and filters and because it includes a series of components known as filters that are linked together by pipes. A pipe is a method of passing information from one part to another.Each filter will function autonomously, accepting data in a specific format and outputting data in the same format for the subsequent filter to process. There is no need to understand how adjacent filters function in order to use them.

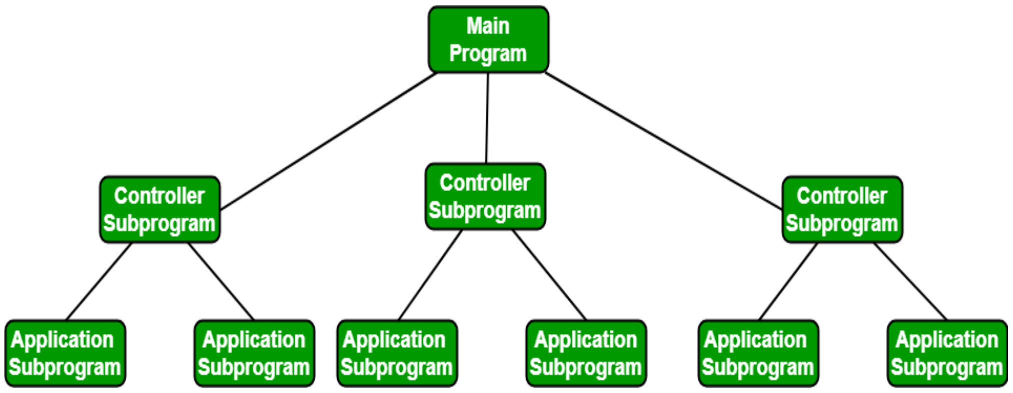
It is said that a data flow is considered batch sequential if it degenerates into a single line of transformations. A sequence of sequential components are applied to the batch of data in this design.

Call and Return architectures:

It's utilized to make a flexible and extensible app. There are a plethora of subgenres within this style. Two of them are discussed in detail below.

Using a system of remote procedure calls: These parts are shown in a networked main programmed or sub-program architecture that runs on several machines simultaneously.

Architectures of the Core and the Subprograms: The primary programmer structure is broken down into a control hierarchy consisting of a variety of subprograms or functions. The main software is broken up into several smaller modules that can call on external resources.



V. Object oriented architecture

Data and the necessary actions for manipulating that data are packaged up in the system's components. Message transmission facilitates coordination and communication between parts.

Layered architecture:

Several layers are specified, each of which is responsible for a certain set of operations. The actions performed by each successive layer will become increasingly closer to the machine instruction set. Components in the outer layer will handle user interface operations, while components in the inner levels will handle OS interfacing (communication and coordination with OS) and intermediate layers to utility services and application software activities..

What Sets Architectural Patterns Apart From Design Patterns:

A common misconception among developers is that architectural styles and patterns are interchangeable. Unfortunately, not all programmers can differentiate between architecture and design patterns. The article's purpose is to provide a concise explanation of such distinctions.

According to the Microsoft Developer Network, there is no difference between architectural styles and patterns. But how is it possible? Pattern refers to a recurring ornamental motif, whereas style refers to a particular way of accomplishing things. These definitions, however, illustrate contradictory ideas. The language used in software engineering has to be more precise in order to convey the intended meaning. What then, is the difference between the terms, and how do we tell them apart?

5.1 Architectural Style

Our architectural style reveals how we structure our code or, at a higher level of abstraction, how our system would seem from a bird's-eye view. In addition, we pay special attention to the interplay between our system's layers and modules as we develop its architectural style. There are many distinct architectural styles to choose from, and it is also possible to create a hybrid design by combining elements from two or more different styles. Here are some examples of various architectural styles:

Layered architecture, pipelines and filters, and component-based architecture are some examples of architectural styles that can be structured.Such as Implicit invocation, Asynchronous Message Passing, and the Publish-Subscribe Model of Message Delivery.
Examples of distributed systems include those that are service-oriented, peer-to-peer in nature, based on an object request broker architecture, or utilize cloud computing.
Role-based, blackboard, and database-centric are all examples of shared-memory approaches.
Forms of system adaptability include the microkernel approach, language reflection, and domain-specific coding paradigms.

5.2 Patterns in Architecture

An example of how a solution may be implemented to address a persistent issue is provided by the architectural pattern. In other words, it reveals the interplay between parts of a code or the code itself. In addition, the architectural pattern describes our system's architectural style and fixes any problems with that style. The way I see it, the best method to put our architectural style into practise is through the definition of architectural patterns. Consider the question, "How do we, in our architectural style, divide off the UI from the data module?" How do we include an external module into our existing infrastructure? In our client-server setup, how many wheels will we have? Some common architectural patterns include the following: micro services, message bus, service requester/consumer, model-view-controller, model-view-view model, microkernel, n-tier, domain-driven design, and presentation-abstraction-control.

5.3 Design Patterns

Design patterns are a collection of lessons learned and best practices utilized by software engineers to address common issues encountered when creating software. In their 1994 book "Design Pattern - Elements of reusable object-oriented software," the Gang of Four (GOF, referring to Eric Gamma, Richard Helm, Ralf Johnson, and John Vlissides) proposed that design patterns are founded on two key tenets of OO design:

1. Focus on the user experience instead than the underlying technology.Make use of composition rather than inheritance when creating objects.
2. They also demonstrated that there are a total of 23 patterns in the design patterns collection, which can be broken down into the following groups:

1. Creational design patterns:

Make it possible to generate objects while concealing the code responsible for doing so. That's why it's best practise to avoid creating new objects by using the "New" keyword; doing so frees up developers to choose which objects they need to construct based on the specifics of a given scenario. The generative design patterns are as follows:
The goal of the abstract factory design is to allow for the creation of objects without requiring knowledge of their classes.

To implement the singleton design, just one instance of a service is created and made available for all callers.
With the Builder Pattern, you can use the same construction to generate different representations by decoupling it from the representation.
The Prototype Design Pattern allows for copies to be made with minimal impact on speed or memory use. Thus, the original object's framework serves as the basis for the copy.

5.4 Structural patterns:

Related to the composition of classes and objects. Examples of structural design patterns are:

An adapter pattern combines the features of two different interfaces.

The bridge design allows you to separate an abstraction from the code that implements it.
The filter pattern, also known as the criterion pattern, allows for the decoupled chaining of logical operations to filter a collection of objects according to a variety of criteria.
If you want to handle many objects as though they were one, the composite pattern is for you. It uses a tree structure to represent both individual levels and entire hierarchies by composing things in a certain way.
Using the Decorator pattern, you may extend an object's capabilities without changing its fundamental design.

The Façade pattern is used to create a standardized front end for several underlying interfaces.
It conceals the system's intricacies and gives the client an interface to use to interact with the system.
The goal of the flyweight pattern is to improve efficiency and memory use by decreasing the total number of objects generated. Assists in reusing similar things by saving them and making a new one if none of the stored objects are a match.

The proxy pattern involves substituting one item for another for the sake of regulating access to the latter. The object's functionality is exposed to the outside world through a unique interface.

5.5 Behavioral patterns:

Interactions between things are a major focus of behavioural patterns. Behavior patterns include the following:

The Responsibility Pattern establishes a sequence of objects to process a request. In this structure, the request's originator and its intended recipient are separated into two distinct groups, each of which is responsible for handling a certain kind of request.
In the command pattern, a request is encapsulated in a command object and delivered to an invoker object.

The Interpreter Pattern: an approach to assessing linguistic expressiveness and grammar. The process entails putting in place a way to provide a context-specific interpretation. SQL parsing, symbol processing engines, and other similar systems all make use of this pattern.
The iterator pattern enables sequential access to the components of a collection object without exposing their internal representation.
The complexity of communication between various objects or classes may be minimised with the help of the Mediator design. It allows for loose coupling between classes, which makes code maintenance simpler, and offers a mediator class that handles class-to-class communication by default.
This pattern, known as the "Memento," is employed when it is necessary to return an object to an earlier state.When two items have a one-to-many connection, such as when one object is updated and its dependent objects must be alerted immediately, the Observer design is employed.
The class's actions can be modified depending on the current state using the state pattern.
Avoiding null references is made easier with the aid of the null object pattern.

The Strategy pattern allows for dynamic modifications to a class's behavior or logic.
When using the template pattern, an abstract class will provide a set of predefined templates for implementing its methods. Subclasses may change the method's implementation as necessary, but they must call it in the same way that the abstract class specifies.
The purpose of the Visitor pattern is to alter the algorithm that a class of elements uses to perform an action.

The bottom line:

The design aesthetic is like looking down from a 1,000,000 foot vantage point. An extremely high-level representation of the system's design. It also demonstrates the application's high-level components and their interplay. Contrarily, architectural patterns have a significant effect on the horizontal and vertical scales of system implementation. In the end, the design patterns are put to use during software development to address specific problems. Because design patterns focus on a narrower aspect of code implementation, such as creating and communicating amongst objects, their influence is less pervasive than that of architectural patterns.

Conclusion:

Currently available software development models point us in the proper way and aid us in accomplishing our goals. However, in the future, developers will need to govern some aspects of software development models, such as the usage of graphical symbols that are readily understood by users and which also define the problem which is to be solved. Moreover, in the near future, we will need to construct new software development models due to shifting client demands and the rising push to innovate.

References:

[1]. K. K. Aggarwal, Yogesh Singh Software Engineering 3rd Edition.
[2]. Karlm, "Software Lifecycle Models', KTH, 2006.
[3]. Vanshika Rastogi 'Software Development Life Cycle Models-Comparison, Consequences' ISNN: 0975-9646 International Journal of Computer Science and Information Technologies, Vol. 6 (1) , 2015, 168-172
[4]. Sanjana Taya "Comparative Analysis of Software Development Life" ISSN: 2229-4333 (print), ISSN: 0976-8491(online), vol.2ISSUE 4, Oct-Dec2011
[5]. Nabil Mohammed Ali Munassar And A.Govardhan, "A Comparison between Five Models of Software Engineering" Ijcsi International Journal of Computer Science Issues, Vol7, Issue 5, Sep, 2010.
[6]. Comparative Analysis of Different types of Models in Software Development Life Cycle-Ms. Shikha
Maheshwari, Prof. Dinesh Ch. Jain