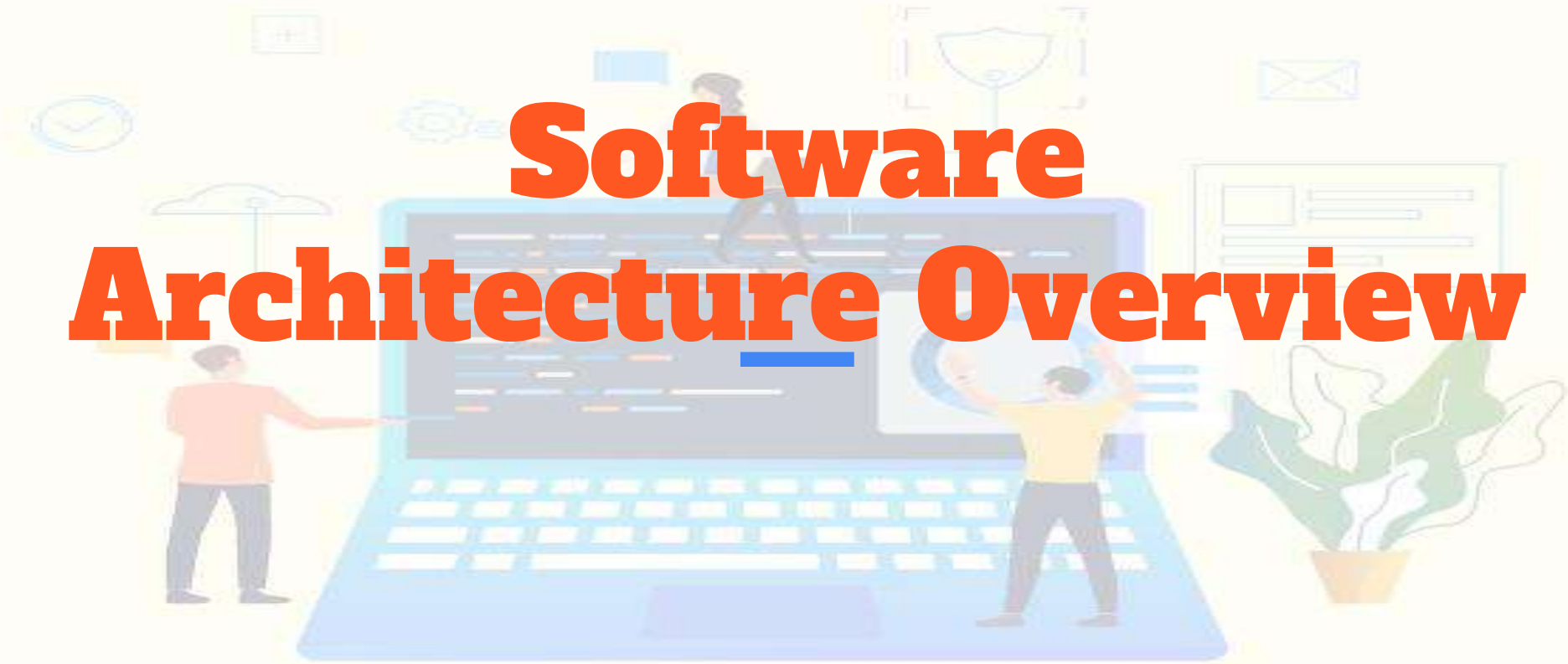
Software Architecture & Design

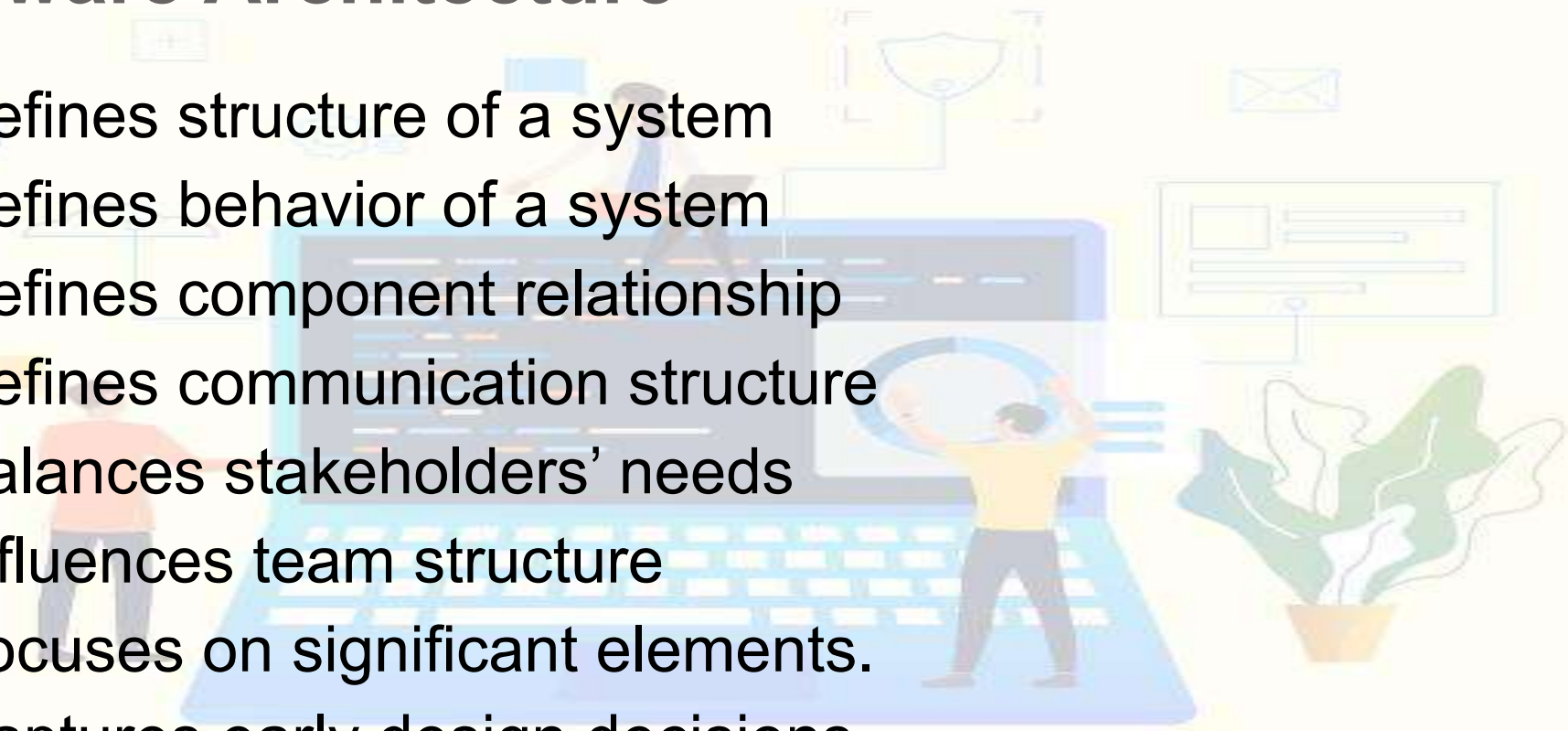
Software Architecture Overview



Software Architecture

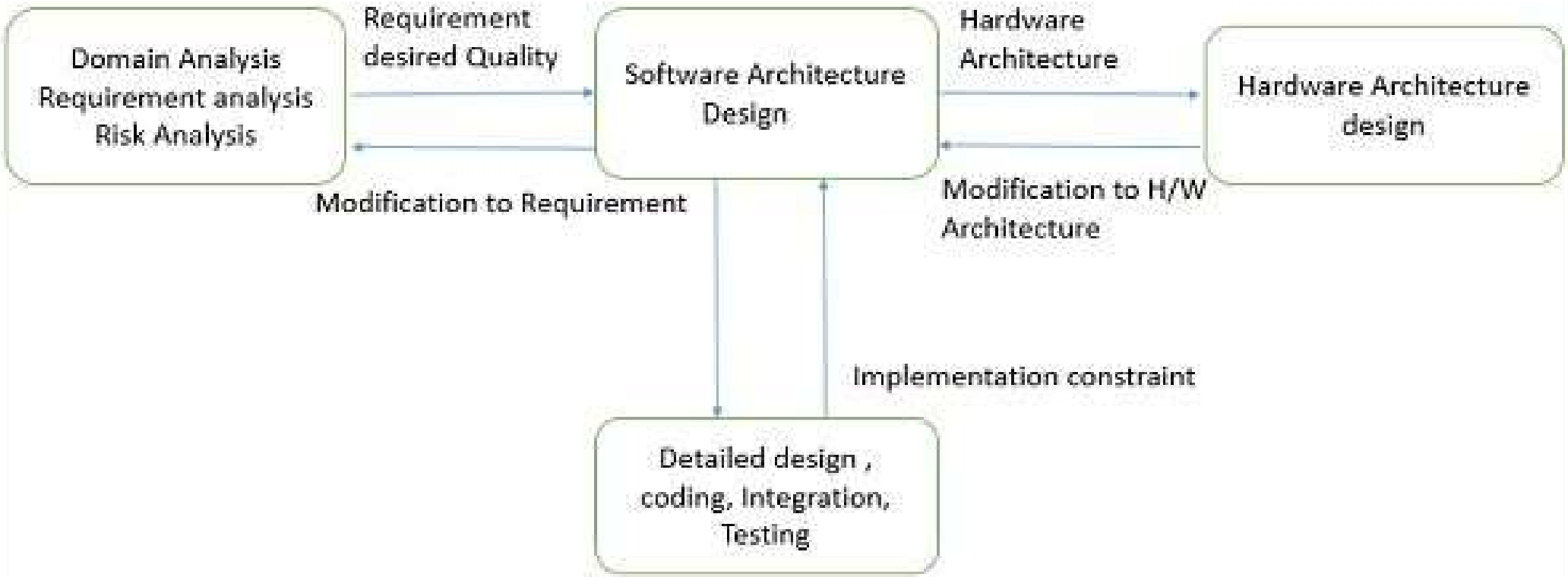
- Serves as a ***blueprint for a system*** – provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.
- Defines a ***structured solution*** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security

Software Architecture



- Defines structure of a system
- Defines behavior of a system
- Defines component relationship
- Defines communication structure
- Balances stakeholders' needs
- Influences team structure
- Focuses on significant elements.
- Captures early design decisions.

Software Architecture



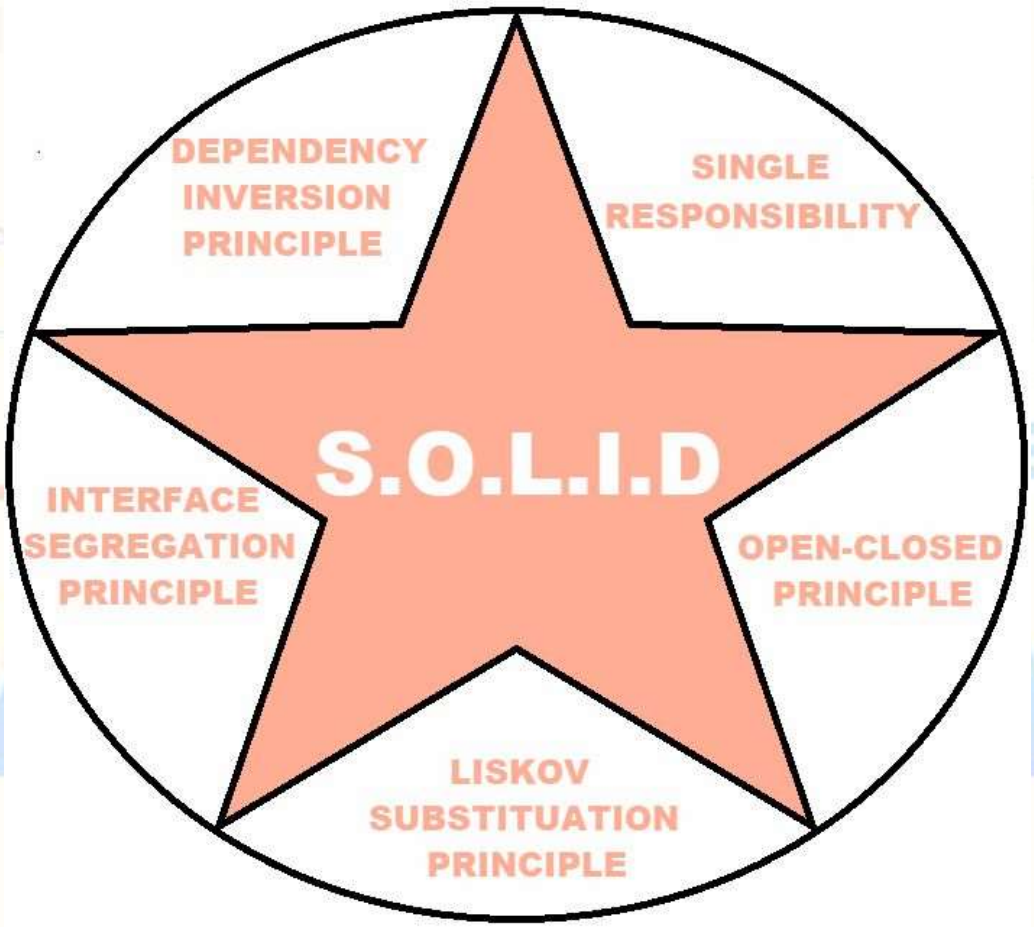
Software Architecture Goal

- Primary goal: ***Identify requirements that affect the structure of the application***
  - Reduce the business risks associated with building a technical solution and builds a bridge between business and technical requirements

Software Architecture Goal

- Expose the structure of the system, but hide its implementation details
- Realize all the use-cases and scenarios
- Try to address the requirements of various stakeholders
- Handle both functional and quality requirements
- Reduce the goal of ownership and improve the organization’s market position
- Improve quality and functionality offered by the system
- Improve external confidence in either the organization or system

Software Architecture Principles



Software Architecture Principles

- **Single Responsibility** – each services should have a single objective
- **Open-Closed Principle** – software modules should be independent and expandable
- **Liskov Substitution Principle** – independent services should be able to communicate and substitute each other

Software Architecture Principles

- **Interface Segregation Principle** – software should be divided into such microservices there should not be any redundancies
- **Dependency Inversion Principle** – Higher-levels modules should not be depending on low-lower-level modules and changes in higher level will not affect to lower level

Software Architecture Quality Attributes

Categories:

- Design Quality
- Run-time Quality
- System Quality
- User Quality
- Architecture Quality
- Non-runtime Quality
- Business Quality

Design Quality

- ***Conceptual Integrity*** – Defines the consistency and coherence of the overall design. This includes the way components or modules are designed
- ***Maintainability*** – ability of the system to undergo changes with a degree of ease
- ***Reusability*** – defines the capability for components and subsystems to be suitable for use in other applications

Run-time Quality

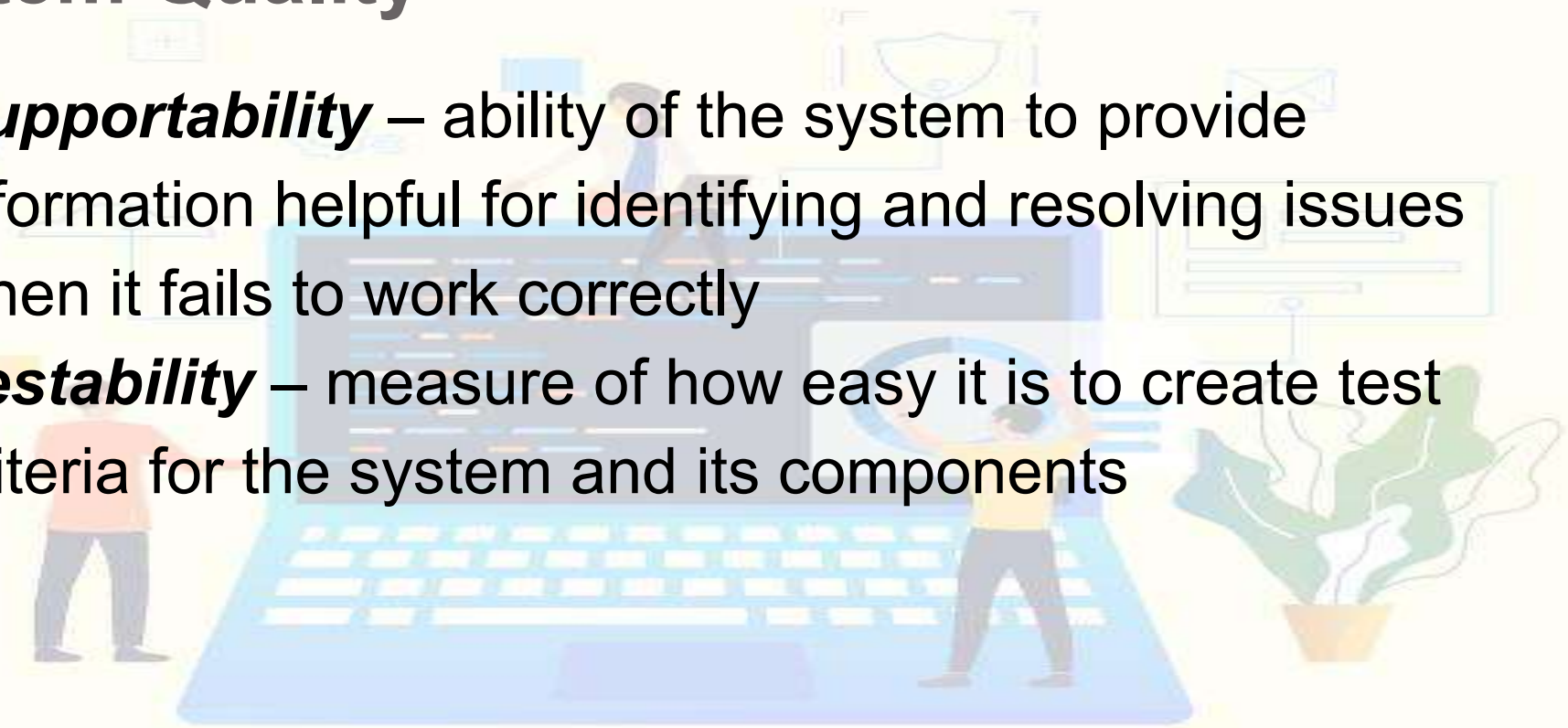
- ***Interoperability*** Ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties
- ***Manageability*** – defines how easy it is for system administrators to manage the application
- ***Reliability*** – ability of a system to remain operational over time

Run-time Quality

- ***Scalability*** – ability of a system to either handle the load increase without impacting the performance of the system or the ability to be readily enlarged
- ***Security*** – capability of a system to prevent malicious or accidental actions outside of the designed usages
- ***Performance*** – indication of the responsiveness of a system to execute any action within a given time interval
- **Availability** – defines the proportion of time that the system is functional and working

System Quality

- ***Supportability*** – ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly
- ***Testability*** – measure of how easy it is to create test criteria for the system and its components



User Quality

- ***Usability*** – defines how well the application meets the requirements of the user and consumer by being intuitive

Architecture Quality

- ***Correctness*** – accountability for satisfying all the requirements of the system

Non-runtime Quality

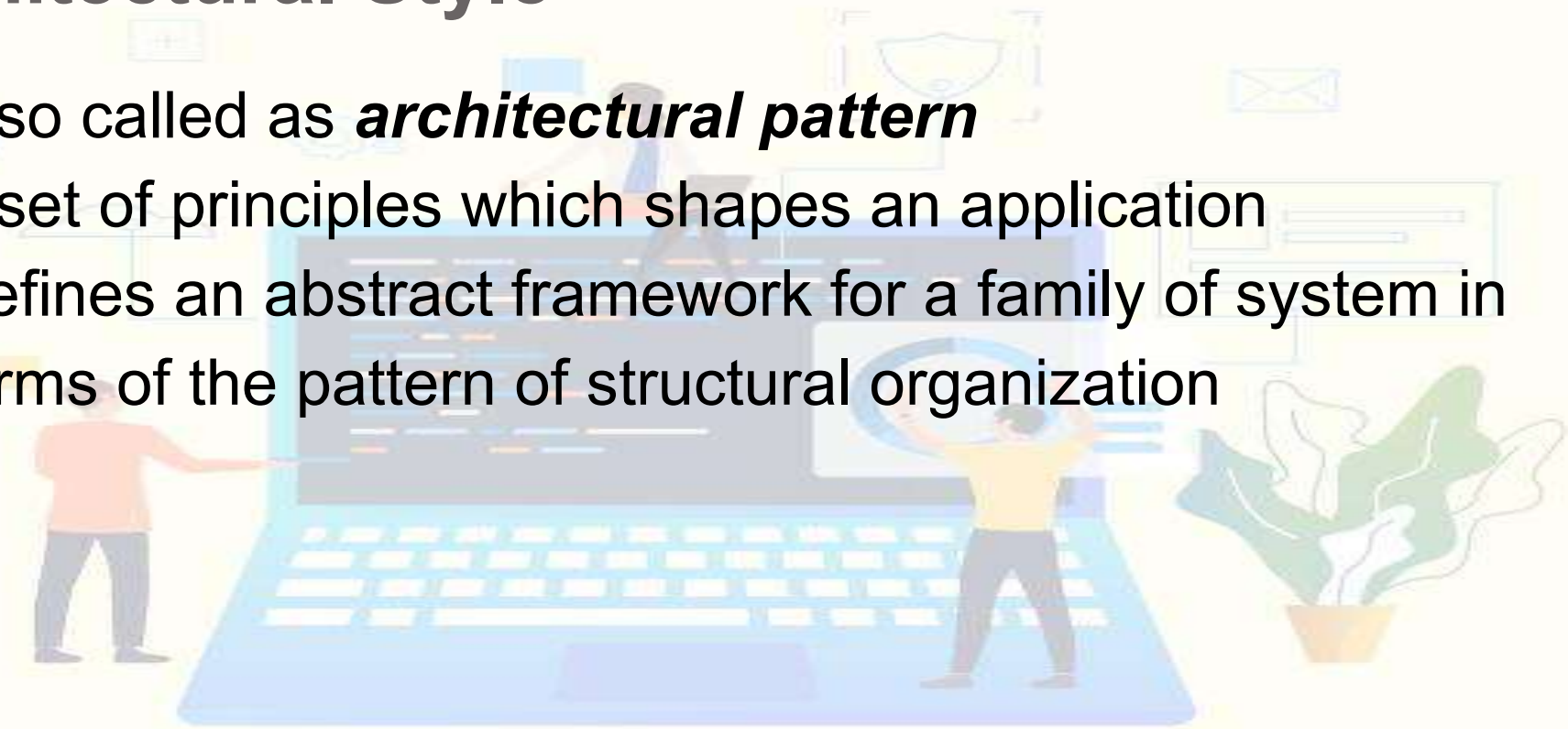
- **Portability** – ability of the system to run under different computing environment
- **Integrality** – ability to make separately developed components of the system work correctly together
- **Modifiability** – ease with which each software system can accommodate changes to its software

Business Quality

- **Portability** – ability of the system to run under different computing environment
- **Integrality** – ability to make separately developed components of the system work correctly together
- **Modifiability** – ease with which each software system can accommodate changes to its software

Architectural Style

- Also called as ***architectural pattern***
- A set of principles which shapes an application
- Defines an abstract framework for a family of system in terms of the pattern of structural organization



Architectural Style

Responsible to:

- Provide a lexicon of components and connectors with rules on how they can be combined
- Improve partitioning and allow the reuse of design by giving solutions to frequently occurring problems
- Describe a particular way to configure a collection of components (a module with well-defined interfaces, reusable, and replaceable) and connectors (communication link between modules)

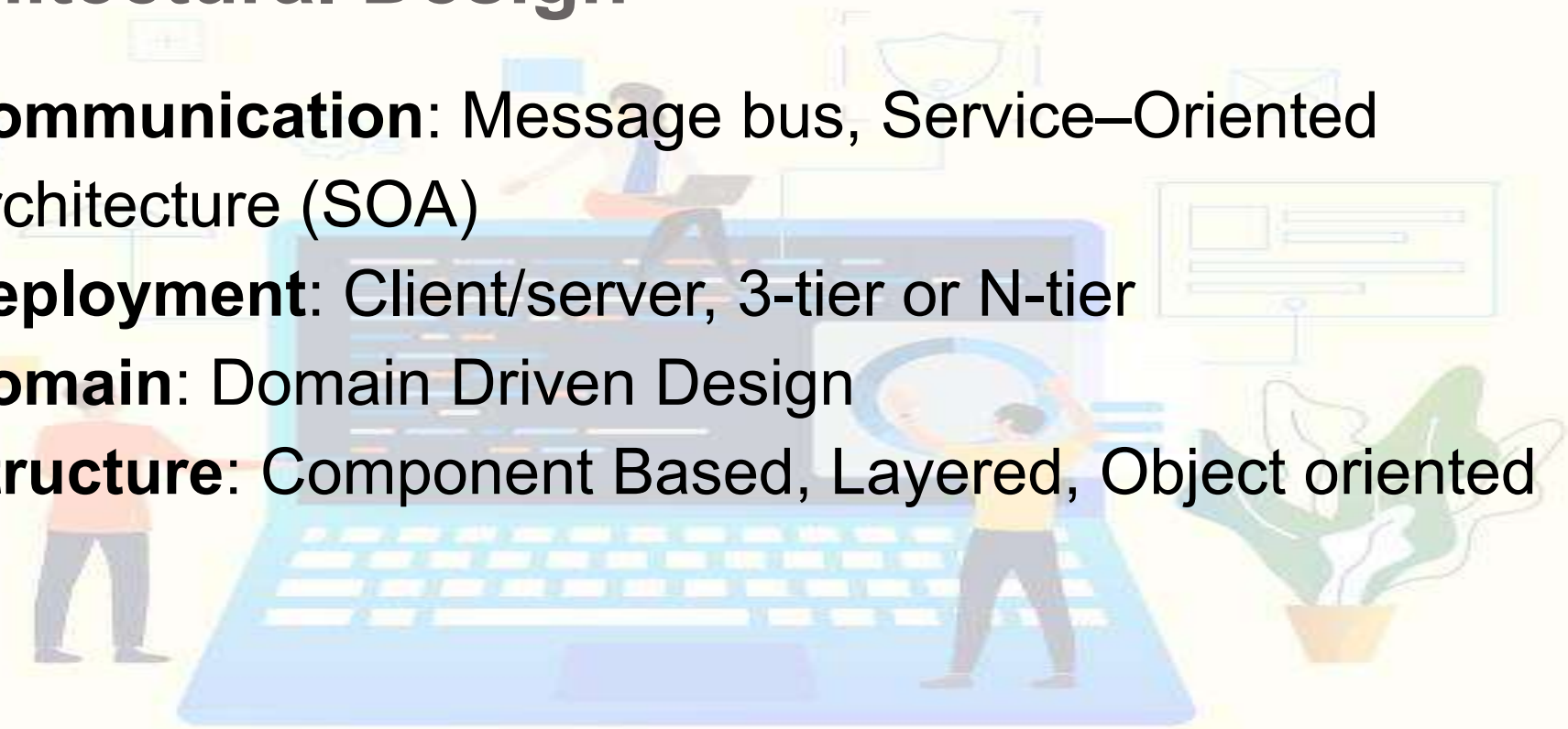
Architectural Style

Describes a system category that encompasses:

- A set of component types which perform a required function by the system
- A set of connectors (subroutine call, remote procedure call, data stream, and socket) that enable communication, coordination, and cooperation among different components
- Semantic constraints which define how components can be integrated to form the system
- A topological layout of the components indicating their runtime interrelationships

Architectural Design

- **Communication**: Message bus, Service–Oriented Architecture (SOA)
- **Deployment**: Client/server, 3-tier or N-tier
- **Domain**: Domain Driven Design
- **Structure**: Component Based, Layered, Object oriented



Architectural Design

Message Bus

- Prescribes use of a software system that can receive and send messages using one or more communication channels

Service–Oriented Architecture (SOA)

- Defines the applications that expose and consume functionality as a service using contracts and messages

Architectural Design

Client-Server

- Separate the system into two applications, where the client makes requests to the server

3-tier / N-tier

- Separates the functionality into separate segments with each segment being a tier located on a physically separate computer

Architectural Design

Domain Driven Design

- Focused on modeling a business domain and defining business objects based on entities within the business domain

Component Based

- Breakdown the application design into reusable functional or logical components that expose well-defined communication interfaces

Architectural Design

Layered

- Divide the concerns of the application into stacked groups (layers)

Object Oriented

- Based on the division of responsibilities of an application or system into objects, each containing the data and the behavior relevant to the object

Architectural Design Process

- Focuses on the ***decomposition of a system into different components*** and their interactions to ***satisfy functional and nonfunctional requirements***
- ***Key Inputs:***
  - The ***requirements*** produced by the analysis tasks
  - The ***hardware architecture***
- ***Output:***
  - ***Architectural Description*** document

Architectural Design Process Steps

STEP 1: Understand the Problem

- This is the most crucial step because it affects the quality of the design that follows
- Without a ***clear understanding of the problem***, it is not possible to ***create an effective solution***
- Many software projects and products are considered failures because they did not actually solve a valid business problem or have a recognizable return on investment (ROI)

Architectural Design Process Steps

STEP 2: Identify Design Elements & their Relationships

- Build a baseline for defining the boundaries and context of the system
- Decomposition of the system into its main components based on functional requirements *(using a **Design Structure Matrix (DSM)**, which shows the dependencies between design elements without specifying the granularity of the elements)*
- In this step, the first validation of the architecture is done by describing a number of system instances and this step is referred as ***functionality based architectural design***

Architectural Design Process Steps

STEP 3: Evaluate the Architecture Design

- Each quality attribute is given an estimate so in order to gather qualitative measures or quantitative data, the design is evaluated
- It involves evaluating the architecture for conformance to ***architectural quality attributes requirements***
  - If all estimated quality attributes are as per the required standard, the architectural design process is finished.
  - If not, the third phase of software architecture design is entered: architecture transformation
  - If the observed quality attribute does not meet its requirements, then a new design must be created

Architectural Design Process Steps

STEP 4: Transform the Architecture Design

- This step is performed after an evaluation of the architectural design
  - The architectural design must be changed until it completely satisfies the quality attribute requirements
- It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality
- A design is transformed by applying design operators, styles, or patterns
  - For transformation, take the existing design and apply design operator such as decomposition, replication, compression, abstraction, and resource sharing

Architectural Design Process Steps

STEP 4: Transform the Architecture Design

- The design is again evaluated and the same process is repeated multiple times if necessary and even performed recursively
- The transformations (i.e. quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively

Key Architecture Principles

- **Build to Change Instead of Building to Last**

  Consider how the application may need to change over time to address new requirements and challenges, and build in the flexibility to support this

Key Architecture Principles

- **Reduce Risk and Model to Analyze**

  Use design tools, visualizations, modeling systems such as UML to capture requirements and design decisions. The impacts can also be analyzed. Do not formalize the model to the extent that it suppresses the capability to iterate and adapt the design easily.

Key Architecture Principles

- **Use Models and Visualizations as a Communication and Collaboration Tool**

  Efficient communication of the design, the decisions, and ongoing changes to the design is critical to good architecture. Use models, views, and other visualizations of the architecture to communicate and share the design efficiently with all the stakeholders. This enables rapid communication of changes to the design.

Key Architecture Principles

- **Use an Incremental and Iterative Approach**

  Start with baseline architecture and then evolve candidate architectures by iterative testing to improve the architecture. Iteratively add details to the design over multiple passes to get the big or right picture and then focus on the details.

Key Design Principles

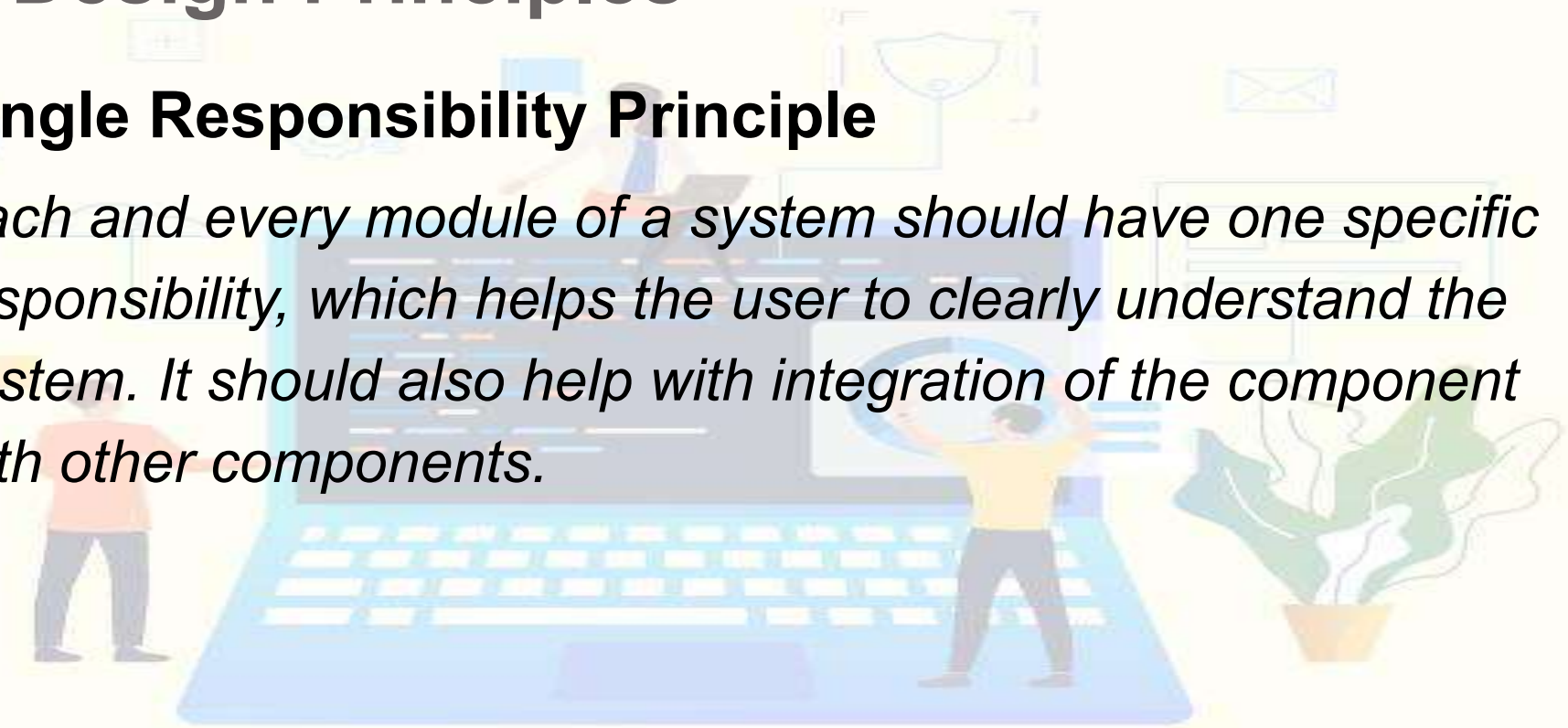
- **Separation of Concerns**

  Divide the components of system into specific features so that there is no overlapping among the components functionality. This will provide high cohesion and low coupling. This approach avoids the interdependency among components of system which helps in maintaining the system easy.

Key Design Principles

- **Single Responsibility Principle**

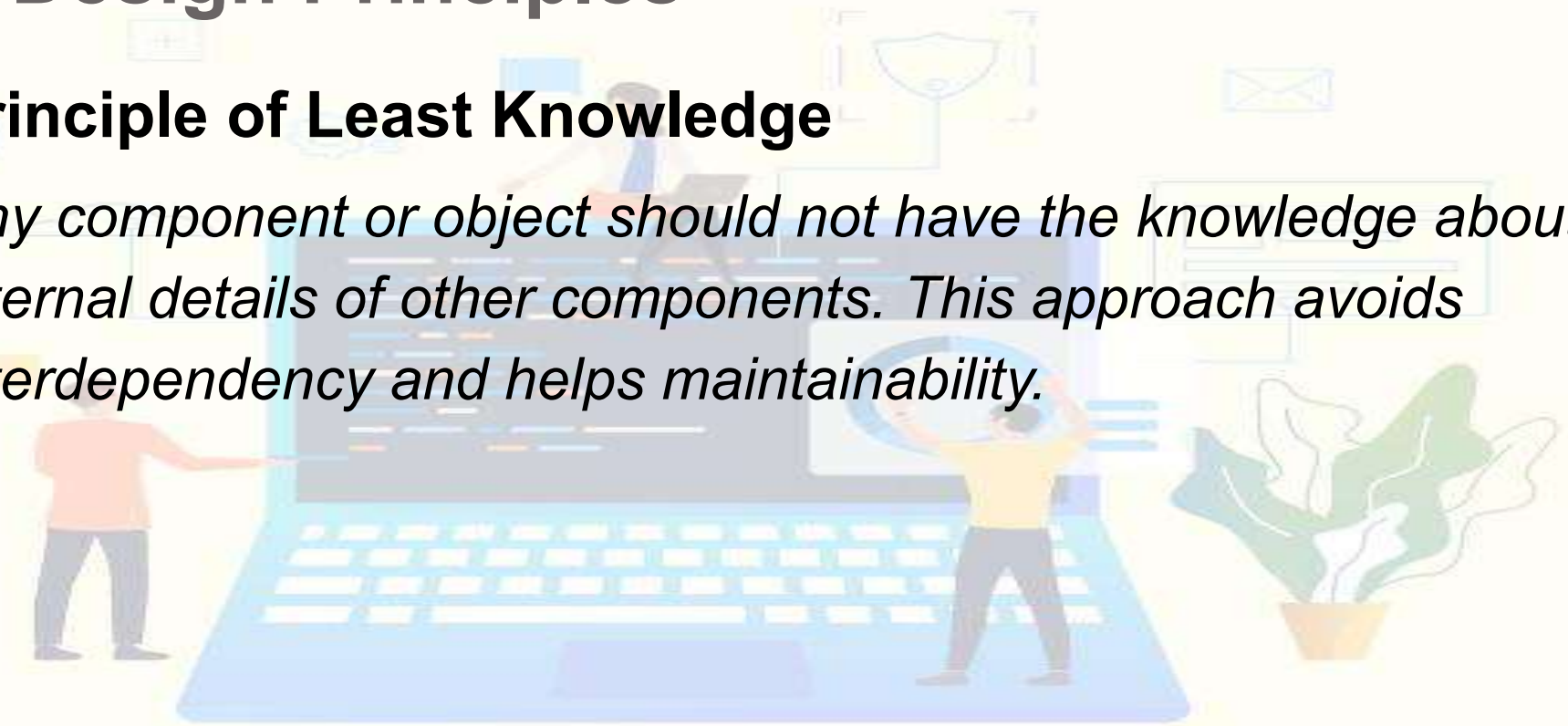
  Each and every module of a system should have one specific responsibility, which helps the user to clearly understand the system. It should also help with integration of the component with other components.



Key Design Principles

- **Principle of Least Knowledge**

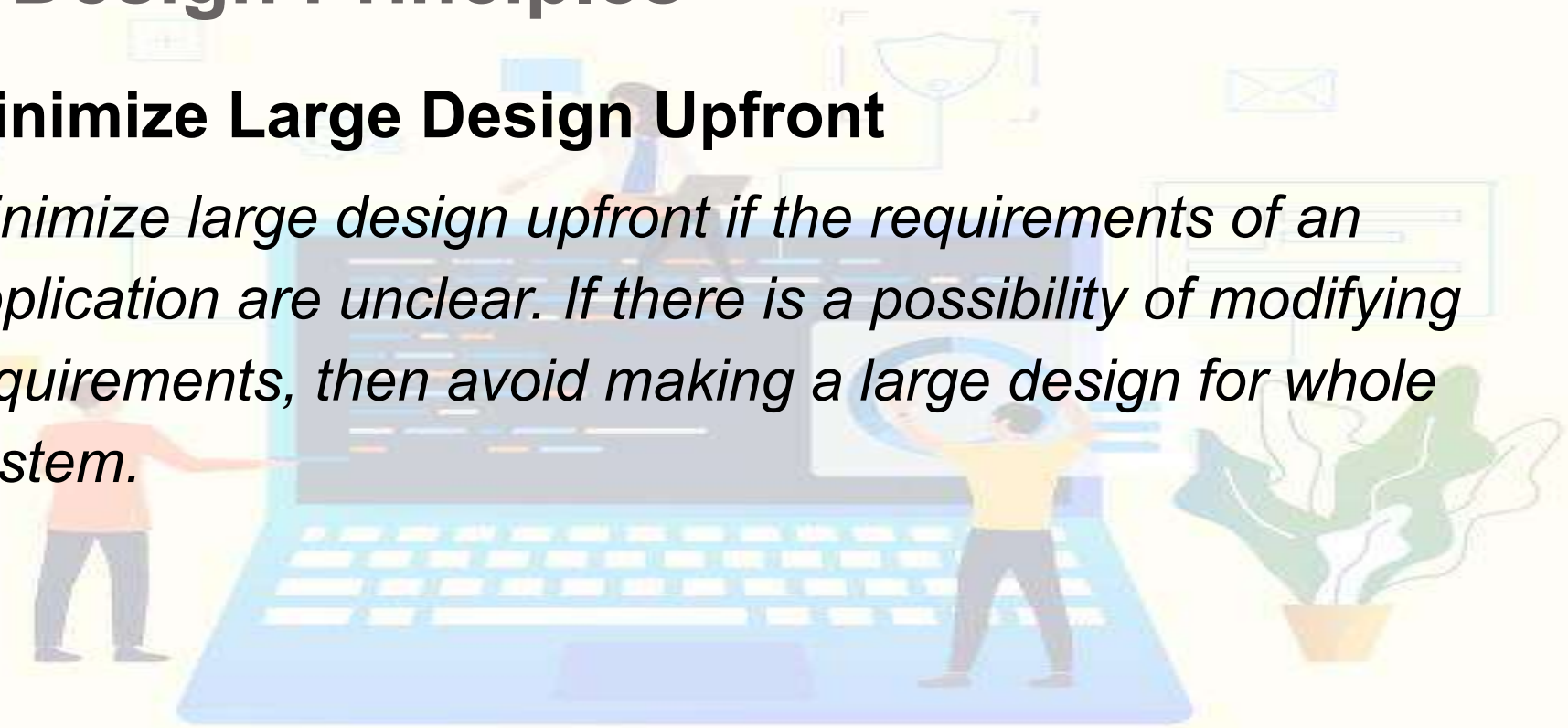
  Any component or object should not have the knowledge about internal details of other components. This approach avoids interdependency and helps maintainability.



Key Design Principles

- **Minimize Large Design Upfront**

  Minimize large design upfront if the requirements of an application are unclear. If there is a possibility of modifying requirements, then avoid making a large design for whole system.



Key Design Principles

- **Do not Repeat the Functionality**

  Do not repeat functionality specifies that functionality of components should not to be repeated and hence a piece of code should be implemented in one component only. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.

Key Design Principles

- **Prefer Composition over Inheritance while Reusing the Functionality**

  Inheritance creates dependency between children and parent classes and hence it blocks the free use of the child classes. In contrast, the composition provides a great level of freedom and reduces the inheritance hierarchies.

Key Design Principles

- **Identify Components and Group them in Logical Layers**

  Identity components and the area of concern that are needed in system to satisfy the requirements. Then group these related components in a logical layer, which will help the user to understand the structure of the system at a high level. Avoid mixing components of different type of concerns in same layer.

Key Design Principles

- **Define the Communication Protocol between Layers**

  Understand how components will communicate with each other which requires a complete knowledge of deployment scenarios and the production environment.

Key Design Principles

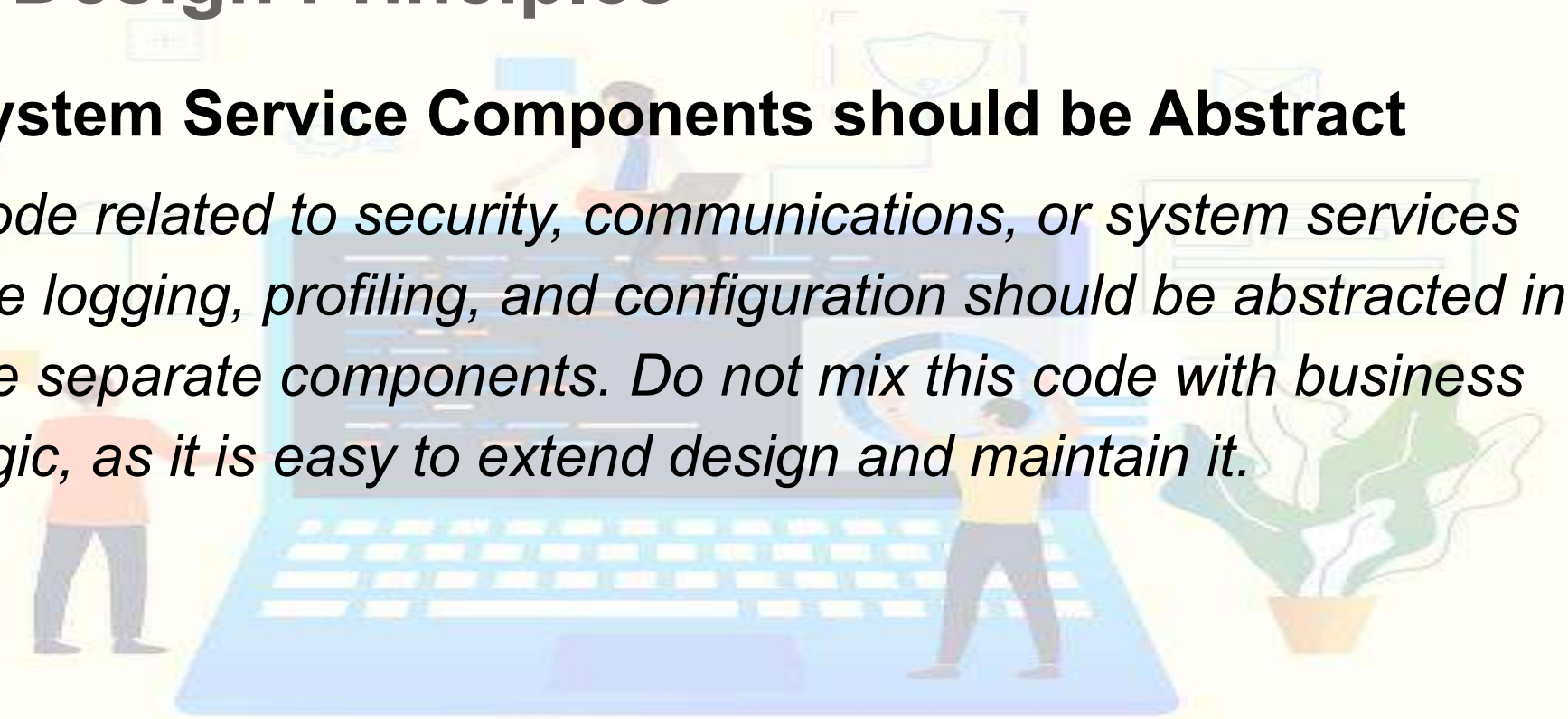
- **Define Data Format for a Layer**

  Various components will interact with each other through data format. Do not mix the data formats so that applications are easy to implement, extend, and maintain. Try to keep data format same for a layer, so that various components need not code/decode the data while communicating with each other. It reduces a processing overhead.

Key Design Principles

- **System Service Components should be Abstract**

  Code related to security, communications, or system services like logging, profiling, and configuration should be abstracted in the separate components. Do not mix this code with business logic, as it is easy to extend design and maintain it.



Key Design Principles

- **Design Exceptions and Exception Handling Mechanism**

  Defining exceptions in advance, helps the components to manage errors or unwanted situation in an elegant manner. The exception management will be same throughout the system.

Key Design Principles

- **Naming Conventions**

  Naming conventions should be defined in advance. They provide a consistent model that helps the users to understand the system easily. It is easier for team members to validate code written by others, and hence will increase the maintainability.

Software Architecture Model

- ***Software architecture*** involves the ***high level structure*** of software system abstraction, by using ***decomposition*** and ***composition,*** with architectural style and quality attributes
- A software architecture design must conform to the ***major functionality*** and ***performance requirements*** of the system, as well as satisfy the n***on-functional requirements*** such as reliability, scalability, portability, and availability

Software Architecture Model

- A software architecture must describe its group of ***components***, their ***connections***, ***interactions*** among them and ***deployment*** configuration of all components that can be defined using:
  - **UML (Unified Modeling Language)**
  - **Architecture View Model (4+1 view model)**
  - **ADL (Architecture Description Language)**

Unified Modeling Language (UML)

- One of object-oriented solutions used in software modeling and design
- A pictorial language used to make software blueprints that was created by Object Management Group (OMG)
- Serves as a standard for software requirement analysis and design documents which are the basis for developing a software

Unified Modeling Language (UML)

- A general purpose visual modeling language to visualize, specify, construct, and document a software system
- The elements are like components which can be associated in different ways to make a complete UML picture, which is known as a ***diagram***
- Categories of UML diagrams:
  - **Structural Diagrams**
  - **Behavioral Diagrams**

UML Structural Diagrams

- Represent the static aspects of a system which forms the main structure and is therefore stable
- These static parts are represented by classes, interfaces, objects, components, & nodes

Class diagram	**Deployment diagram**
Object diagram	**Package diagram**
Component diagram	**Composite structure**

UML Structural Diagrams

Class Diagram

- Represents the object orientation of a system
- Shows how classes are statically related

Object Diagram

- Represents a set of objects and their relationships at runtime and also represent the static view of the system

UML Structural Diagrams

Component Diagram

- Describes all the components, their interrelationship, interactions and interface of the system

Deployment Diagram

- Represents a set of nodes and their relationships and these nodes are physical entities where the components are deployed

UML Structural Diagrams

Package Diagram

- Describes the package structure and organization. Covers classes in the package and packages within another package

Composite Structure

- Describes inner structure of component including all classes, interfaces of the component, etc

UML Behavioral Diagrams

- Basically capture the dynamic aspect of a system
- Dynamic aspects are basically the changing/moving parts of a system

Use case diagram	**Activity diagram**
Sequence diagram	**Interaction overview**
Communication diagram	**Time sequence diagram**
State chart diagram	

UML Behavioral Diagrams

Use Case Diagram

- Describes the relationships among the functionalities and their internal/external controllers(*actors*)

Activity Diagram

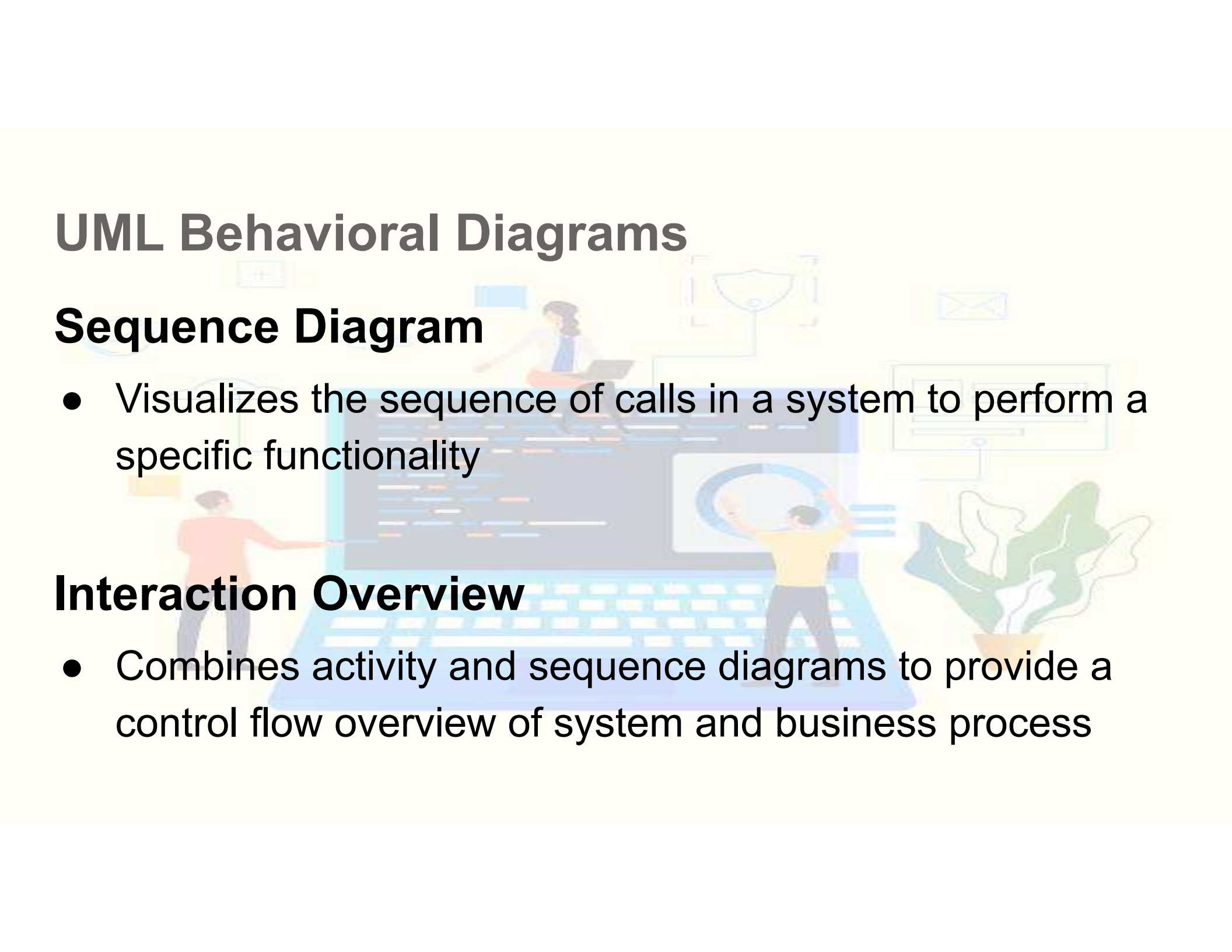
- Describes the flow of control in a system
- Consists of activities and links
- The flow can be sequential, concurrent, or branched

UML Behavioral Diagrams

State Machine Diagram / State Chart

- Represents the event driven state change of a system
- Basically describes the state change of a class, interface, etc
- Used to visualize the reaction of a system by internal/external factors.

UML Behavioral Diagrams



Sequence Diagram

- Visualizes the sequence of calls in a system to perform a specific functionality

Interaction Overview

- Combines activity and sequence diagrams to provide a control flow overview of system and business process

UML Behavioral Diagrams

Communication Diagram

- Same as sequence diagram, except that it focuses on the object's role
- Each communication is associated with a sequence order, number plus the past messages

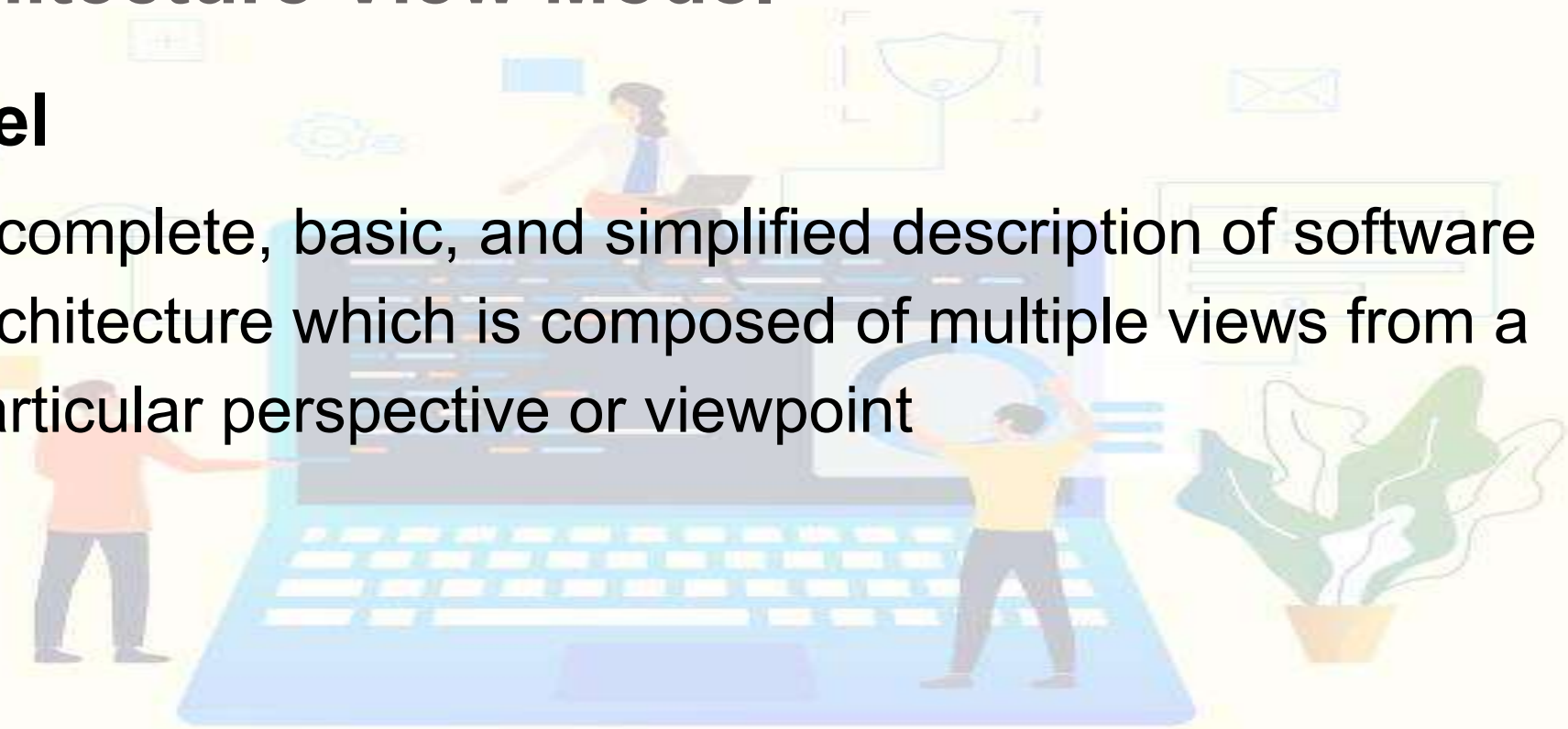
Time Sequenced

- Describes the changes by messages in state, condition & events

Architecture View Model

Model

- A complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint



Architecture View Model

View

- A representation of an entire system from the perspective of a related set of concerns
- Used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers

Architecture View Model: 4+1 View Model

- Designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views
- A multiple view model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders

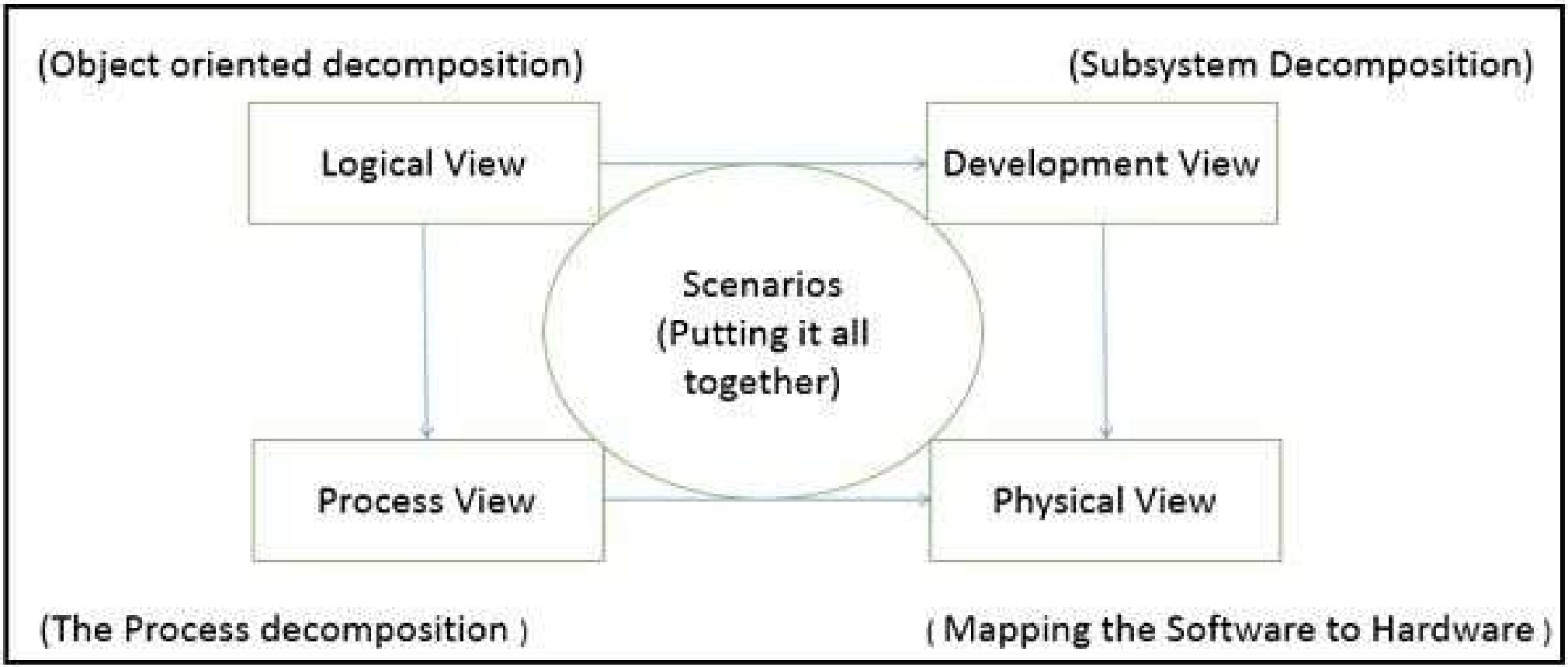
Architecture View Model: 4+1 View Model

- An architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders
- Four Essential Views:
  - Logical view or conceptual view
  - Process view
  - Physical view
  - Development view

4+1 View Model: Use Case View

- This view model can be extended by adding one more view called ***use case view*** or ***scenario view*** for end-users or customers of software systems
- It is coherent with other four views and are utilized to illustrate the architecture serving as **“plus one”** view, (4+1) view model
- The use case view has a special significance as it details the high level requirement of a system while other views details – how those requirements are realized

4+1 View Model: Five Concurrent Views



4+1 View Model: Logical View

- ***Description***: Shows the component (Object) of system as well as their interaction
- ***Viewer/Stakeholder***: End-User, Analysts and Designer
- ***Consider***: Functional requirements
- ***UML Diagrams***: Class, State, Object, sequence, Communication Diagram

4+1 View Model: Process View

- ***Description***: Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system
- ***Viewer/Stakeholder***: Integrators & developers
- ***Consider***: Non-functional Requirements
- ***UML Diagrams***: Activity Diagram

4+1 View Model: Development View

- ***Description***: Gives building block views of system and describe static organization of the system modules
- ***Viewer/Stakeholder***: Programmer and software project managers
- ***Consider***: Software Module organization (Software management reuse, constraint of tools)
- ***UML Diagrams***: Component, Package diagram

4+1 View Model: Physical View

- ***Description***: Shows the installation, configuration and deployment of software application
- ***Viewer/Stakeholder***: System engineer, operators, system administrators and system installers
- ***Consider***: Non-functional requirement regarding to underlying hardware
- ***UML Diagrams***: Deployment diagram

4+1 View Model: Use Case View

- ***Description***: Shows the design is complete by performing validation and illustration
- ***Viewer/Stakeholder***: All the views of their views and evaluators
- ***Consider***: System Consistency and validity
- ***UML Diagrams***: Use case diagram

Architecture Description Language (ADL)

- A language that provides syntax and semantics for defining a software architecture
- A notation specification which provides features for modeling a software system’s conceptual architecture, distinguished from the system’s implementation
- Must support the architecture components, their connections, interfaces, and configurations which are the building block of architecture description

Architecture Description Language (ADL)

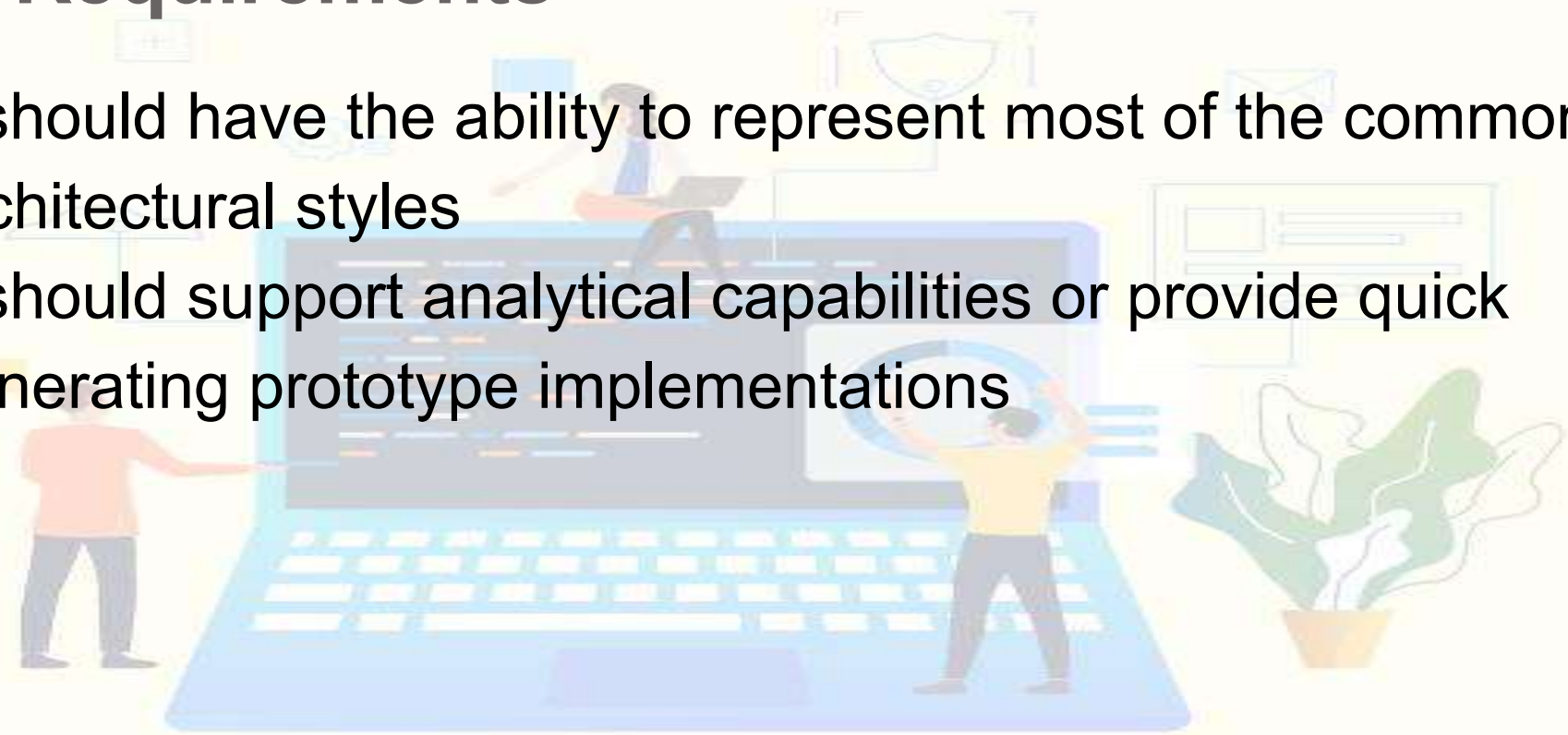
- A form of expression for use in architecture descriptions and provides the ability to decompose components, combine the components, and define the interfaces of components
- A formal specification language, which describes the software features such as processes, threads, data, and sub-programs as well as hardware component such as processors, devices, buses, and memory

ADL Requirements

- It should be appropriate for communicating the architecture to all concerned parties
- It should be suitable for tasks of architecture creation, refinement, and validation
- It should provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL

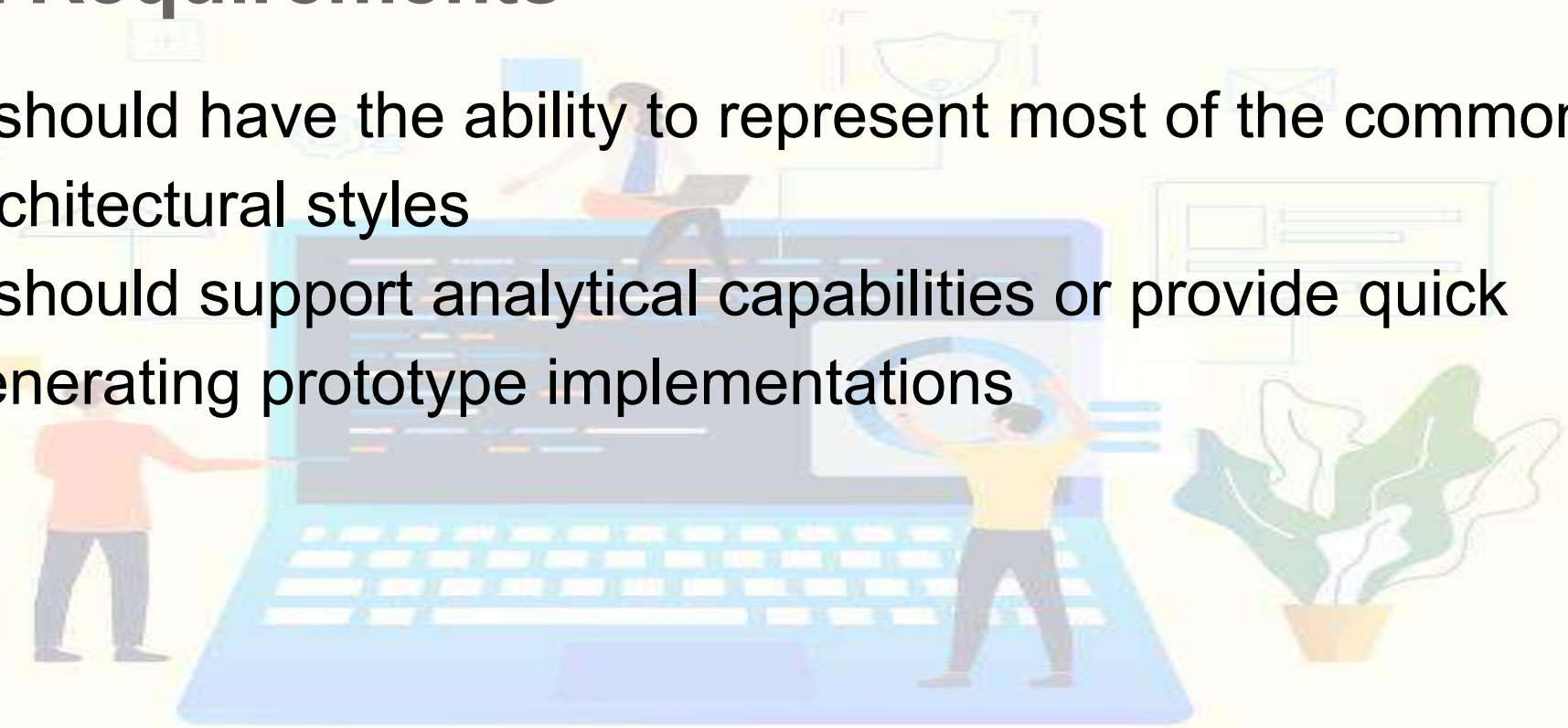
ADL Requirements

- It should have the ability to represent most of the common architectural styles
- It should support analytical capabilities or provide quick generating prototype implementations

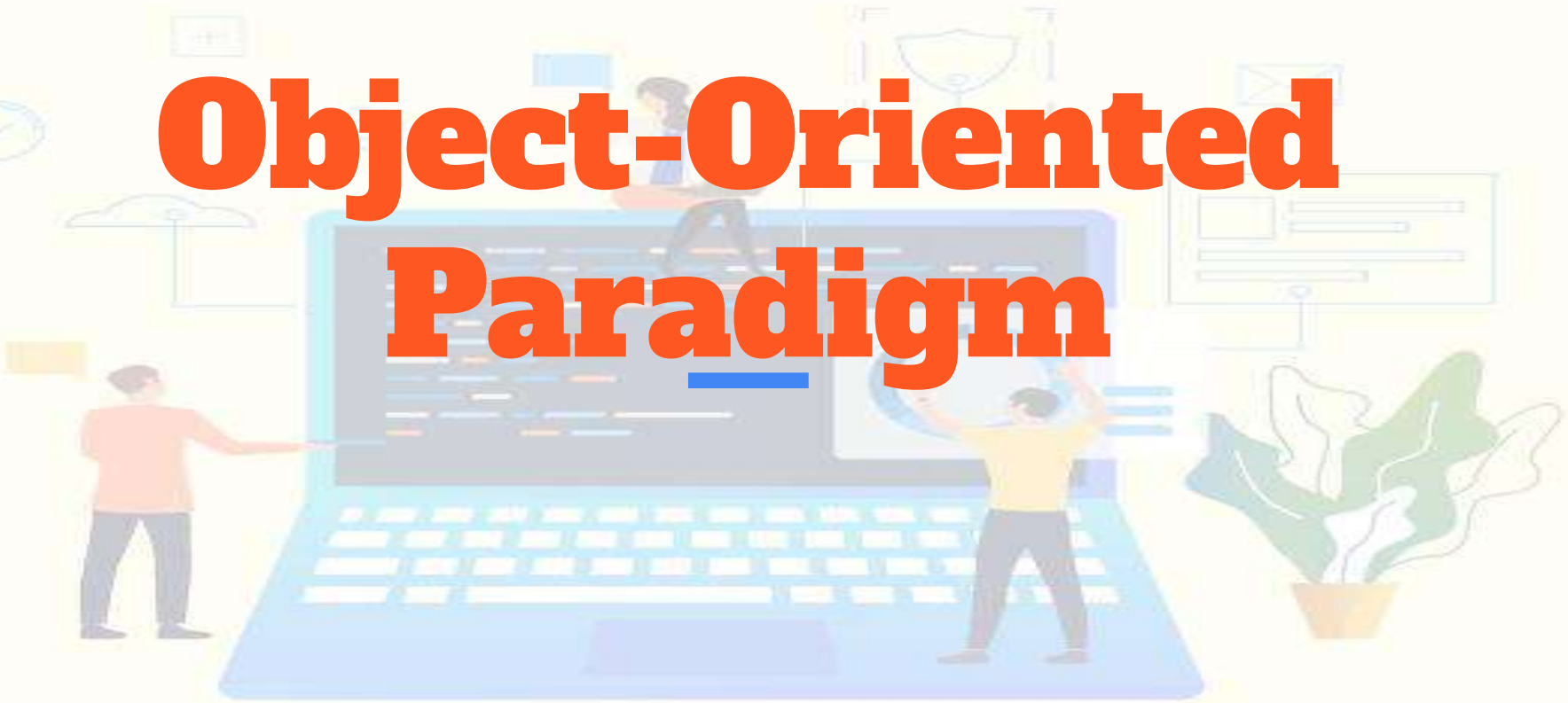


ADL Requirements

- It should have the ability to represent most of the common architectural styles
- It should support analytical capabilities or provide quick generating prototype implementations



Object-Oriented Paradigm



Object-Oriented Paradigm

- OO analysis and design paradigm is the logical result of the wide adoption of OO programming languages
- ***Grady Booch*** *(1980s)* presented a design for the programming language, ***Object Oriented Design***
- ***Coad*** *(1990s)* incorporated behavioral ideas to ***object-oriented methods***
- ***Object Modeling Techniques*** – James Rum Baugh
- ***Object-Oriented Software Engineering*** – Ivar Jacobson

Basic concepts of Object–Oriented Systems

Object

- The real-world elements in an object–oriented environment and can be modeled according to the needs of the application
- An object may have a ***physical existence*** *(e.g. customer, a car, etc.)* or an intangible ***conceptual existence*** *(e.g. a project, a process, etc)*

Basic concepts of Object–Oriented Systems

Object

- Each object has:
  - **Identity** that distinguishes it from other objects in the system
  - **State** that determines characteristic properties of an object as well as values of properties that the object holds
  - **Behavior** that represents externally visible activities performed by an object in terms of changes in its state

Basic concepts of Object–Oriented Systems

Class

- Represents a collection of objects having same characteristic properties that exhibit common behavior
- Gives the blueprint or the description of the objects that can be created from it
- Creation of an object as a member of a class is called **instantiation**

Basic concepts of Object–Oriented Systems

Class

- The constituents of a class are:
  - A ***set of attributes*** for the objects that are to be instantiated from the class *(different objects of a class have some difference in the values of the attributes/class data)*
  - A ***set of operations*** that portray the behavior of the objects of the class *(functions or methods)*

Basic concepts of Object–Oriented Systems

Class

- The constituents of a class are:
  - A ***set of attributes*** for the objects that are to be instantiated from the class *(different objects of a class have some difference in the values of the attributes/class data)*
  - A ***set of operations*** that portray the behavior of the objects of the class *(functions or methods)*

Basic concepts of Object–Oriented Systems

Encapsulation

- The process of binding both attributes and methods together within a class
- Through encapsulation, the internal details of a class can be hidden from outside
- It permits the elements of the class to be accessed from outside only through the interface provided by the class

Basic concepts of Object–Oriented Systems

Polymorphism

- In OO paradigm, it implies using operations in different ways, depending upon the instances they are operating upon
- Allows objects with different internal structures to have a common external interface
- Particularly effective while implementing inheritance

Relationships in OO Paradigm

Message Passing (Dynamic)

- Any application requires a number of objects interacting in a harmonious manner
- Objects in a system may communicate with each other by using message passing
- Example:

  Suppose a system has two objects − obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods

Relationships in OO Paradigm

Composition/Aggregation (Static)

- A relationship among classes by which a class can be made up of any combination of objects of other classes
- It allows objects to be placed directly within the body of other classes
- Referred as a ***"part–of"*** or ***"has–a"*** relationship, with the ability to navigate from the whole to its parts
- An aggregate object is an object that is composed of one or more other objects

Relationships in OO Paradigm

Association (Static)

- A group of links having common structure and common behavior
- Depicts the relationship between objects of one or more classes
- A link can be defined as an instance of an association
- The Degree of an association denotes the number of classes involved in a connection (***unary***, ***binary***, or ***ternary***)

Relationships in OO Paradigm

Inheritance (Static)

- A mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities
- The existing classes are called the ***base classes/parent classes/super-classes***, and the new classes are called the ***derived classes/child classes/subclasses***

Relationships in OO Paradigm

Inheritance (Static)

- The subclass can inherit or derive the attributes and methods of the super-class (es) provided that the super-class allows so
- Besides, the subclass may add its own attributes and methods and may modify any of the superclass methods ***(Overriding)***
- Inheritance defines a ***"is – a" relationship***

Relationships in OO Paradigm

Inheritance (Static)

- The subclass can inherit or derive the attributes and methods of the super-class (es) provided that the super-class allows so
- Besides, the subclass may add its own attributes and methods and may modify any of the superclass methods ***(Overriding)***
- Inheritance defines a ***"is – a" relationship***

Object-Oriented Analysis

- The system requirements are determined, the classes are identified, and the relationships among classes are acknowledged
- The aim is to understand the ***application domain*** and ***specific requirements*** of the system
- The result of this phase is ***requirement specification*** and initial analysis of ***logical structure*** and feasibility of a system
- ***3 analysis techniques*** that are used in conjunction with each other for object-oriented analysis are ***object modeling***, ***dynamic modeling***, and ***functional modeling***

OO Analysis: Object Modeling

- Develops the static structure of the software system in terms of objects.
- Identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects
- Also identifies the main attributes and operations that characterize each class

OO Analysis: Object Modeling

- The process of object modeling can be visualized in the following steps:
  - Identify objects and group into classes
  - Identify the relationships among classes
  - Create a user object model diagram
  - Define a user object attributes
  - Define the operations that should be performed on the classes

OO Analysis: Dynamic Modeling

- The static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined
- Defined as ***"a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world."***

OO Analysis: Dynamic Modeling

- The process of dynamic modeling can be visualized in the following steps:
  - Identify states of each object
  - Identify events and analyze the applicability of actions
  - Construct a dynamic model diagram, comprising of state transition diagrams
  - Express each state in terms of object attributes
  - Validate the state–transition diagrams drawn

OO Analysis: Functional Modeling

- The final component of object-oriented analysis
- Shows the processes that are performed within an object and how the data changes, as it moves between methods. It specifies the meaning of the operations of an object modeling and the actions of a dynamic modeling
- The functional model corresponds to the data flow diagram of traditional structured analysis

OO Analysis: Functional Modeling

- The process of functional modeling can be visualized in the following steps:
  - Identify all the inputs and outputs
  - Construct data flow diagrams showing functional dependencies
  - State the purpose of each function
  - Identify the constraints
  - Specify optimization criteria

Object-Oriented Design (OOD)

- The conceptual model is developed further into an object-oriented model
- In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain
- The main aim of OO design is to develop the ***structural architecture*** of a system

Object-Oriented Design (OOD)

- The stages for object–oriented design can be identified as
  - Defining the context of the system
  - Designing the system architecture
  - Identification of the objects in the system
  - Construction of design models
  - Specification of object interfaces

Stages of OOD: Conceptual Design

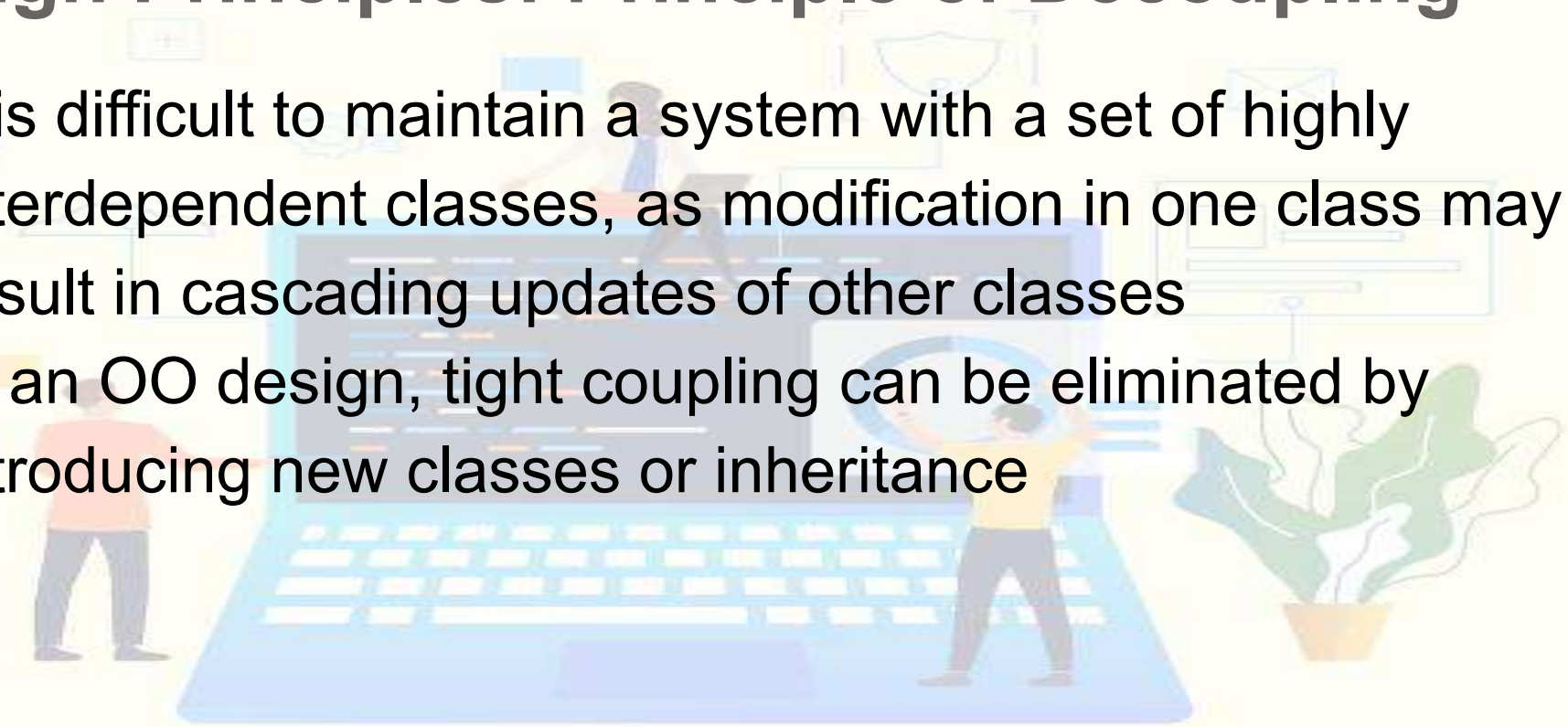
- In this stage, all the classes are identified that are needed for building the system
- Further, specific responsibilities are assigned to each class
- ***Class diagram*** is used to clarify the relationships among classes, and interaction diagram are used to show the flow of events
- It is also known as ***high-level design***

Stages of OOD: Detailed Design

- In this stage, attributes and operations are assigned to each class based on their interaction diagram
- ***State machine diagram*** are developed to describe the further details of design
- ***It is also known as low-level design***
-
-

Design Principles: Principle of Decoupling

- It is difficult to maintain a system with a set of highly interdependent classes, as modification in one class may result in cascading updates of other classes
- In an OO design, tight coupling can be eliminated by introducing new classes or inheritance

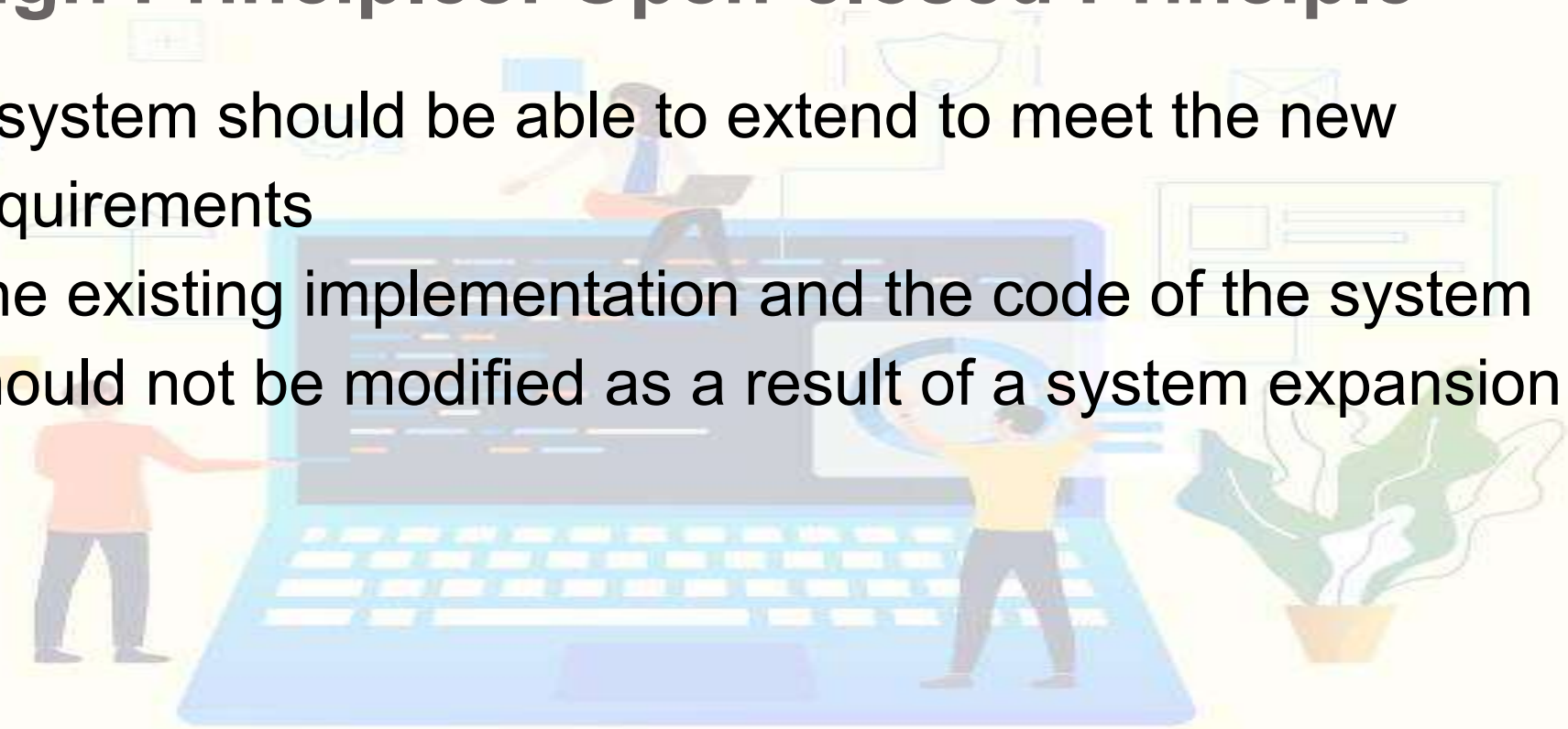


Design Principles: Ensuring Cohesion

- A cohesive class performs a set of closely related functions
- A lack of cohesion means — a class performs unrelated functions, although it does not affect the operation of the whole system
- It makes the entire structure of software hard to manage, expand, maintain, and change

Design Principles: Open-closed Principle

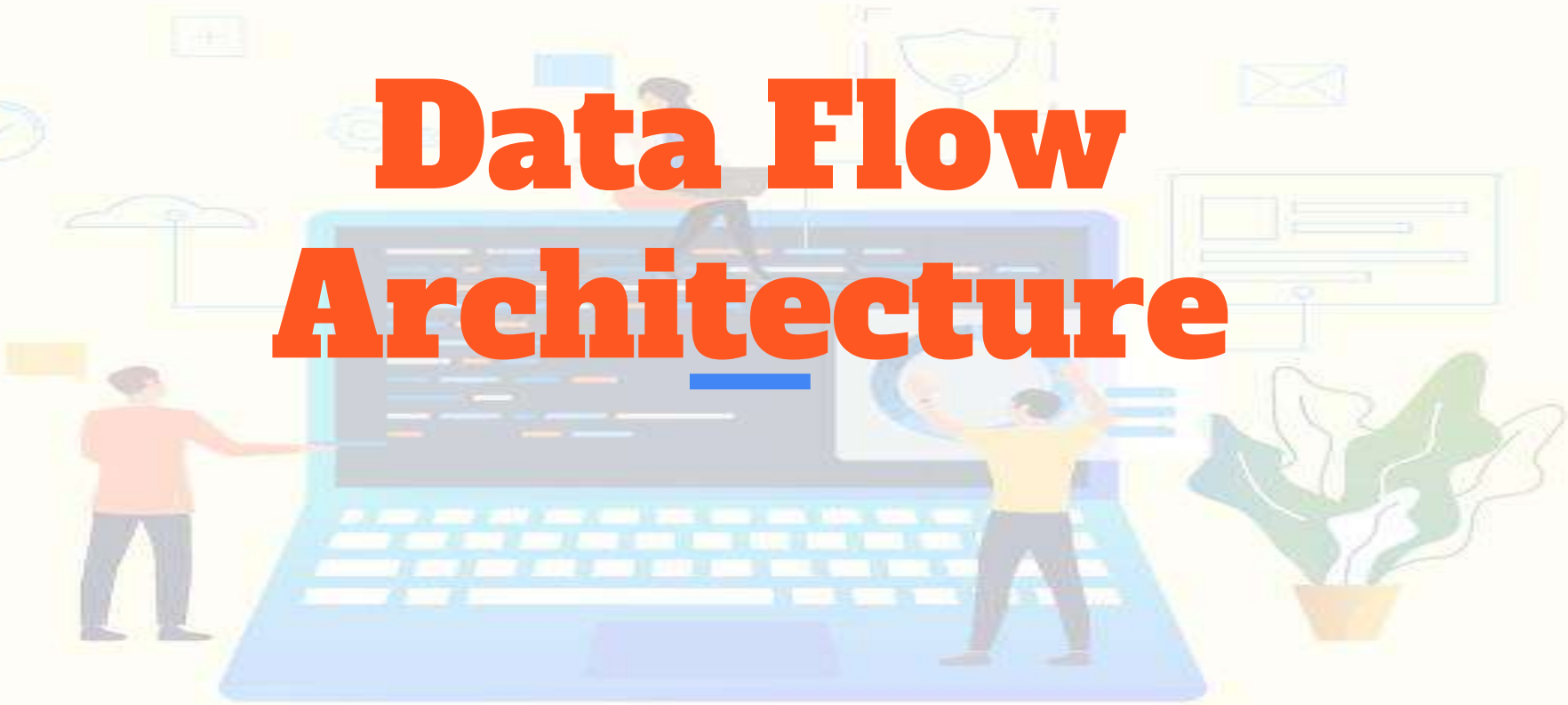
- A system should be able to extend to meet the new requirements
- The existing implementation and the code of the system should not be modified as a result of a system expansion



Design Principles: Open-closed Principle

- In addition, the following guidelines have to be followed in open-closed principle:
  - For each concrete class, separate interface and implementations have to be maintained.
  - In a multithreaded environment, keep the attributes private.
  - Minimize the use of global variables and class variables

Data Flow Architecture



Data Flow Architecture (DFA)

- The whole software system is seen as a series of transformations on consecutive pieces or set of input data, where data, & operations are independent of each other
- In this approach, the data enters into the system and then flows through the modules one at a time until they are assigned to some final destination (output/data store)
- The connections b/w the components or modules may be implemented as I/O stream, I/O buffers, piped, or other types of connections

Data Flow Architecture (DFA)

- ***Main objective***: to achieve the qualities of reuse and modifiability
- Suitable for applications that involve a well-defined series of independent data transformations or computations on orderly defined input and output such as compilers and business data processing applications

Data Flow Architecture (DFA)

- ***Main objective***: to achieve the qualities of reuse and modifiability
- Suitable for applications that involve a well-defined series of independent data transformations or computations on orderly defined input and output such as compilers and business data processing applications
- 3 types of execution sequences b/w modules: ***Batch sequential, Pipe & filter, and Process control***

Batch Sequential

- Classical data processing model, in which a data transformation subsystem can initiate its process only after its previous subsystem is completely through:
  - The flow of data carries a batch of data as a whole from one subsystem to another
  - The communications b/w the modules are conducted through temporary intermediate files which can be removed by successive subsystems
  - It is applicable for those applications where data is batched, & each subsystem reads related input files and writes output files
  - Typical application of this architecture includes business data processing

Batch Sequential

- **Advantages:**
  - Provides simpler divisions on subsystems
  - Each subsystem can be an independent program working on input data and producing output data
- **Disadvantages:**
  - Provides high latency and low throughput
  - Does not provide concurrency and interactive interface
  - External control is required for implementation

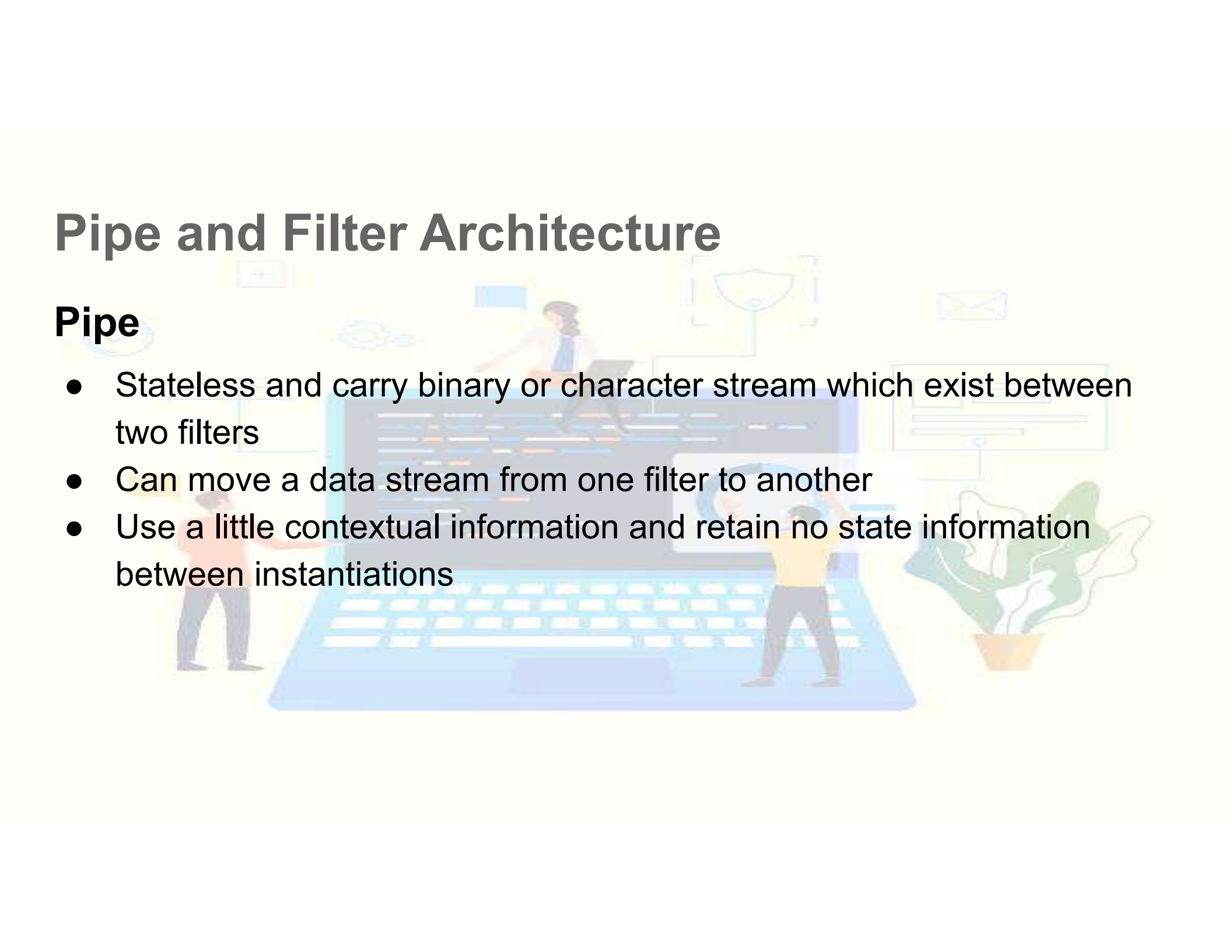
Pipe and Filter Architecture

- Lays emphasis on the incremental transformation of data by successive component
- In this approach, the flow of data is driven by data and the whole system is decomposed into components of data source, filters, pipes, and data sinks
- The connections b/w modules are data stream which is first-in/first-out buffer that can be stream of bytes, characters, or any other type of such kind

Pipe and Filter Architecture

Pipe

- Stateless and carry binary or character stream which exist between two filters
- Can move a data stream from one filter to another
- Use a little contextual information and retain no state information between instantiations



Pipe and Filter Architecture

Filter

- An independent data stream transformer or stream transducers. It transforms the data of the input data stream, processes it, and writes the transformed data stream over a pipe for the next filter to process
- Works in an incremental mode, in which it starts working as soon as data arrives through connected pipe
- 2 types of filters: ***active filter*** & ***passive filter***

Pipe and Filter Architecture

- **Advantages**
  - Provides concurrency and high throughput for excessive data processing
  - Provides reusability and simplifies system maintenance
  - Provides modifiability and low coupling between filters
  - Provides simplicity by offering clear divisions between any two filters connected by pipe
  - Provides flexibility by supporting both sequential and parallel execution

Pipe and Filter Architecture

- **Disadvantages**
  - Not suitable for dynamic interactions
  - A low common denominator is needed for transmission of data in ASCII formats
  - Overhead of data transformation between filters
  - Does not provide a way for filters to cooperatively interact to solve a problem
  - Difficult to configure this architecture dynamically

Process Control Architecture (PCA)

- Type of data flow architecture where data is neither batched sequential nor pipelined stream
- The flow of data comes from a set of variables, which controls the execution of process
- It decomposes the entire system into subsystems or modules and connects them
- Have a ***processing unit*** for changing the process control variables and a ***controller unit*** for calculating the amount of changes

Elements of Controller Unit

- **Controlled Variable:** *Provides values for the underlying system and should be measured by sensors (e.g. speed in cruise control system)*
- **Input Variable:** *Measures an input to the process (e.g. temperature of return air in temperature control system)*
- **Manipulated Variable:** *Value that is adjusted or changed by the controller*
- **Process Definition:** *Includes mechanisms for manipulating some process variables*

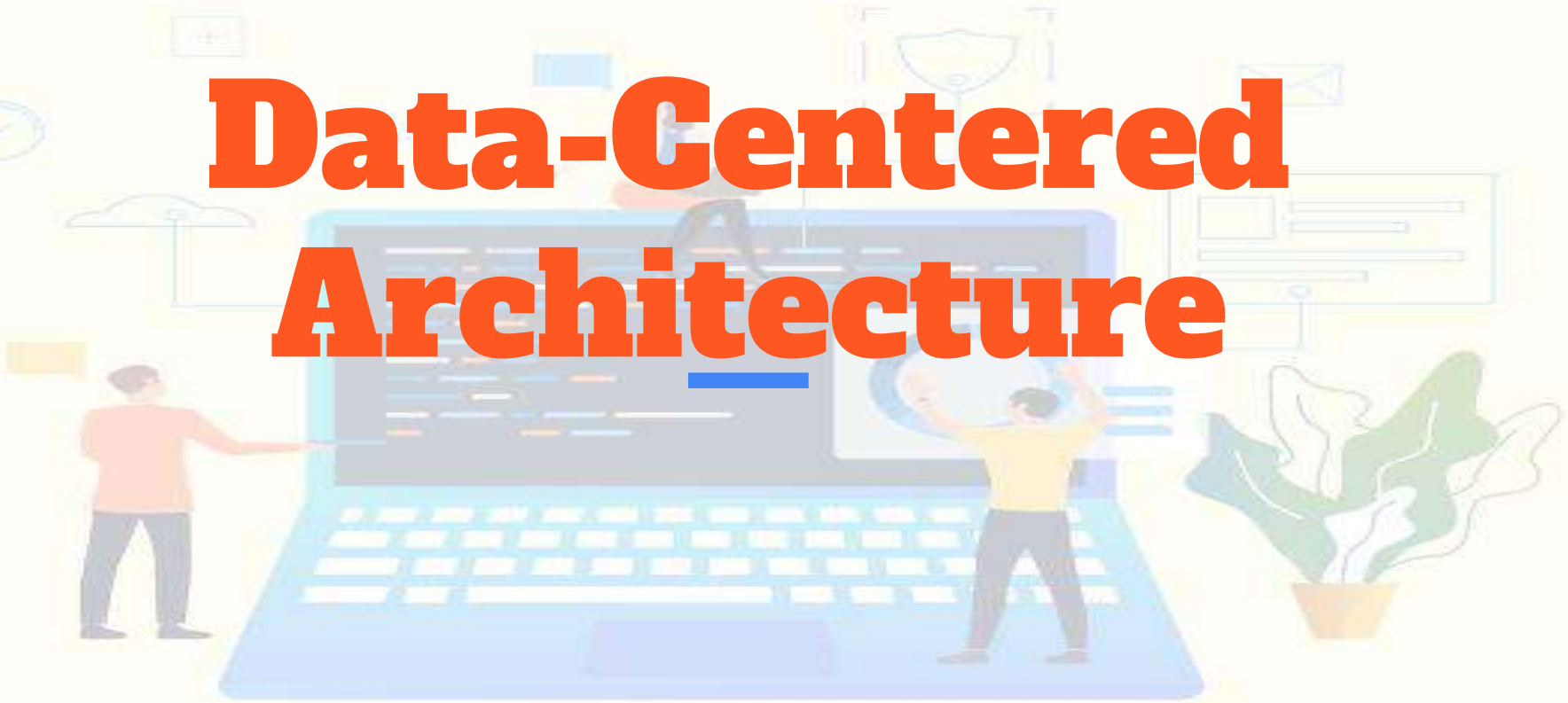
Elements of Controller Unit

- **Sensor:** *Obtains values of process variables pertinent to control and can be used as a feedback reference to recalculate manipulated variables*
- **Set Point:** *The desired value for a controlled variable*
- **Control Algorithm:** *Used for deciding how to manipulate process variables*

PCA Application Areas

- Embedded system software design, where the system is manipulated by process control variable data
- Applications, which aim is to maintain specified properties of the outputs of the process at given reference values
- Applicable for car-cruise control and building temperature control systems
- Real-time system software to control automobile anti-lock brakes, nuclear power plants, etc

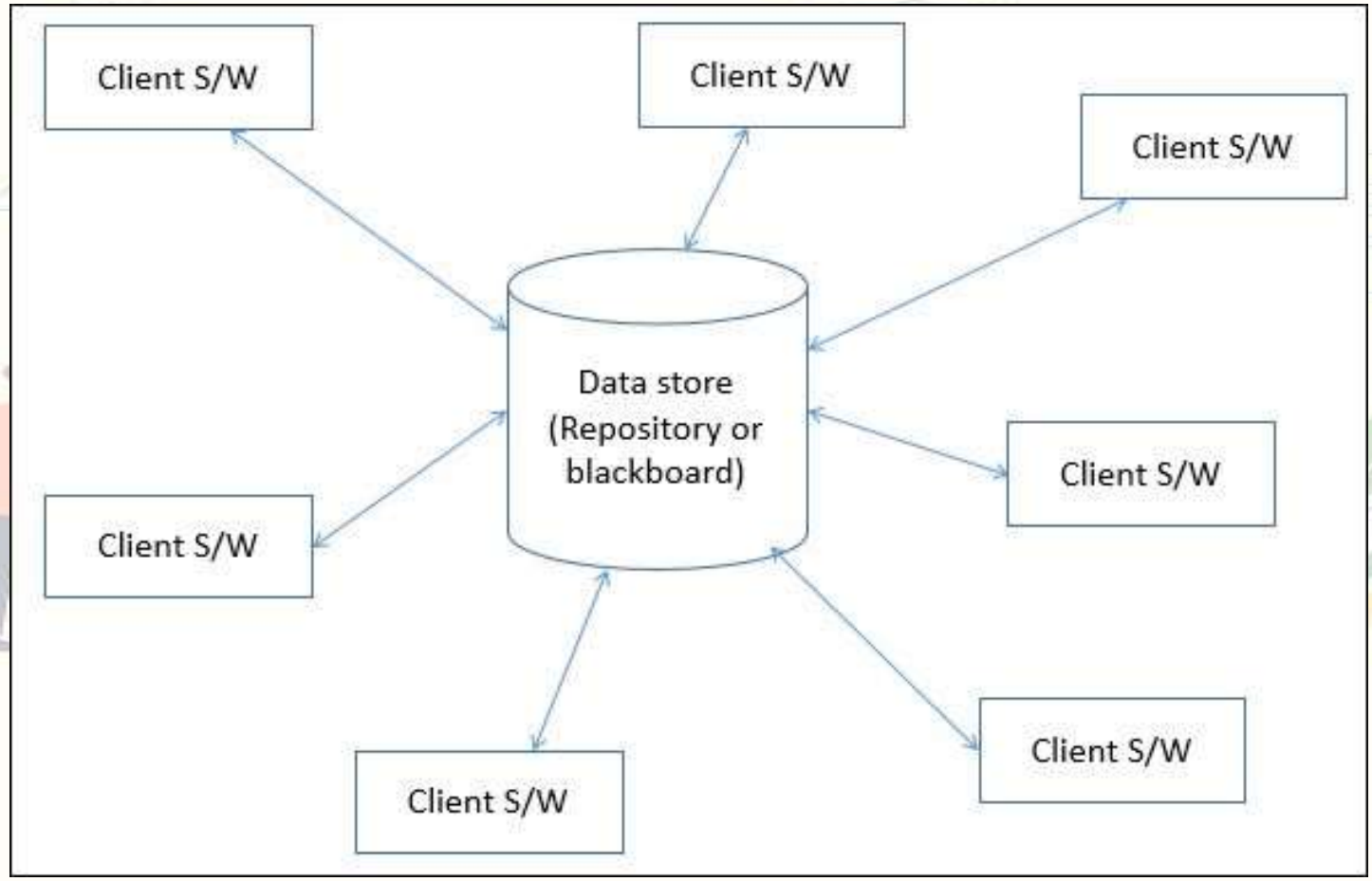
Data-Centered Architecture



Data-Centered Architecture (DCA)

- The data is centralized and accessed frequently by other components, which modify data
- ***Main purpose:*** to achieve integrality of data
- Consists of different components that communicate through shared data repositories and the components access a shared data structure and are relatively independent, in that, they interact only through the data store

Data-Centered Architecture (DCA)



DCA Examples

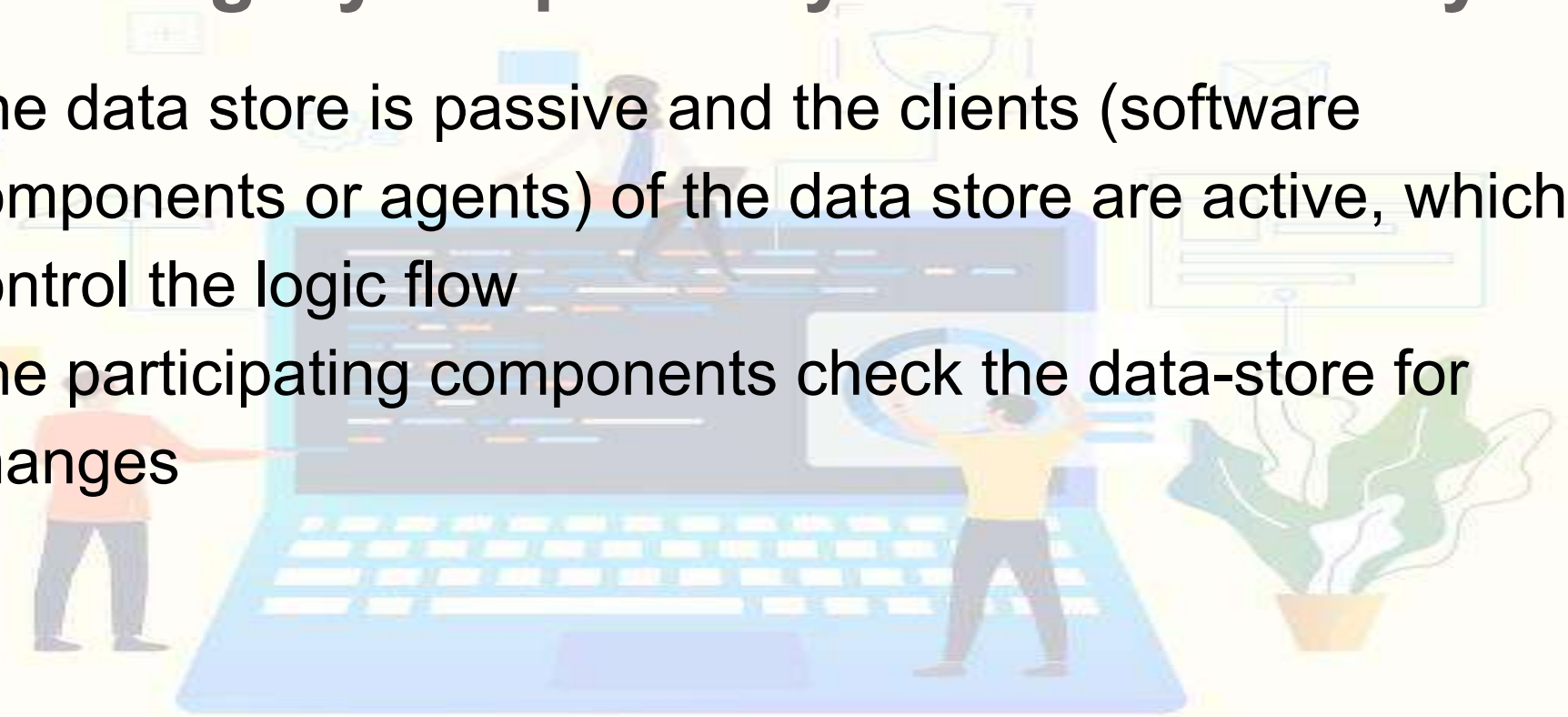
- **Database Architecture,** *which is the common database schema is created with data definition protocol (e.g. a set of related tables with fields and data types in an RDBMS)*
- **Web Architecture,** *which has a common data schema (i.e. meta-structure of the Web) and follows hypermedia data model and processes communicate through the use of shared web-based data services*

DCA Types of Components

- **Central Data Structure / Data Store / Data Repository,** *which is responsible for providing permanent data storage and represents the current state*
- **Data Accessor:** *a collection of independent components that operate on the central data store, perform computations, and might put back the results*

DCA Category: Repository Architecture Style

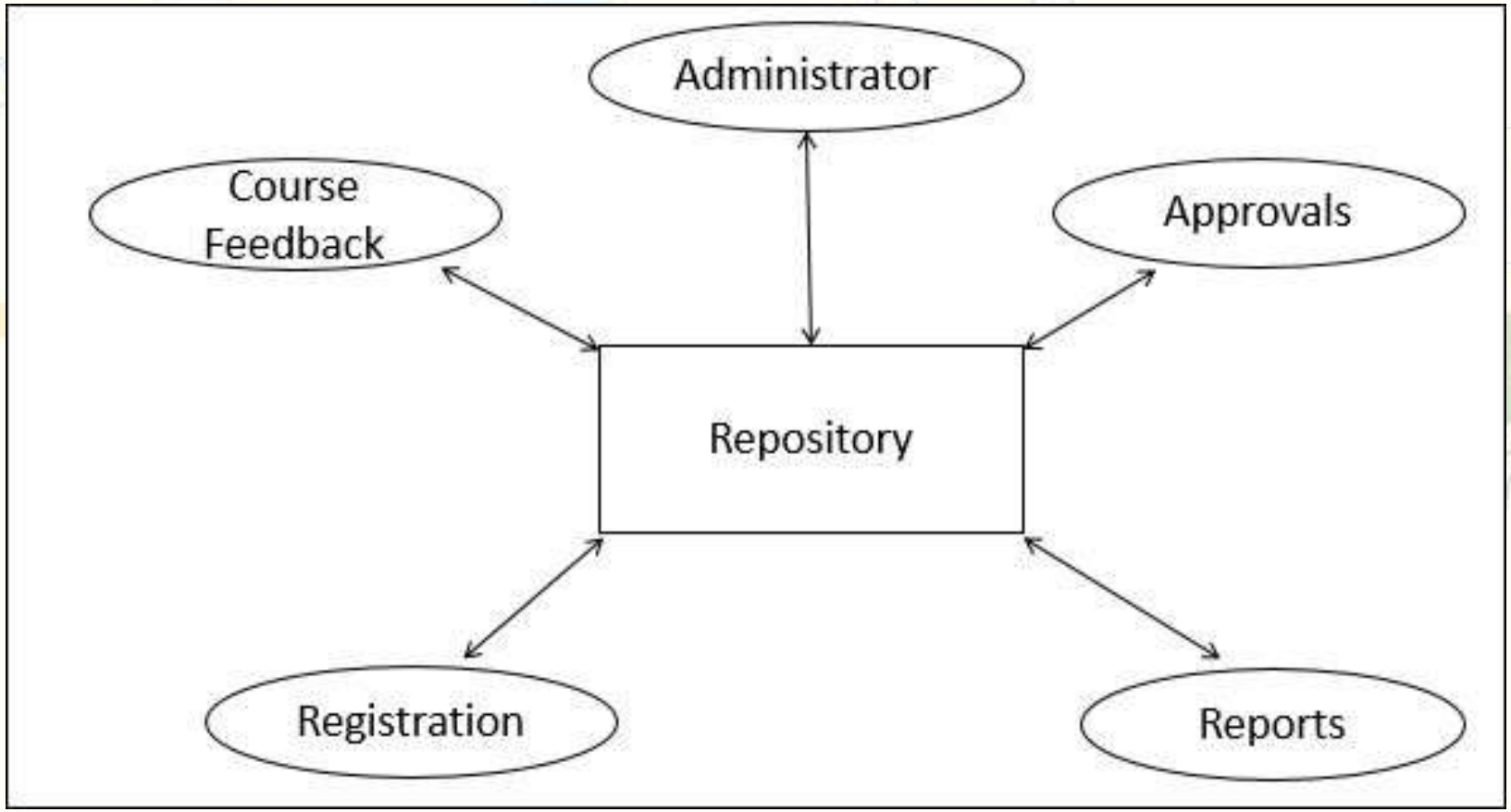
- The data store is passive and the clients (software components or agents) of the data store are active, which control the logic flow
- The participating components check the data-store for changes



About Repository Architecture Style (RAS)

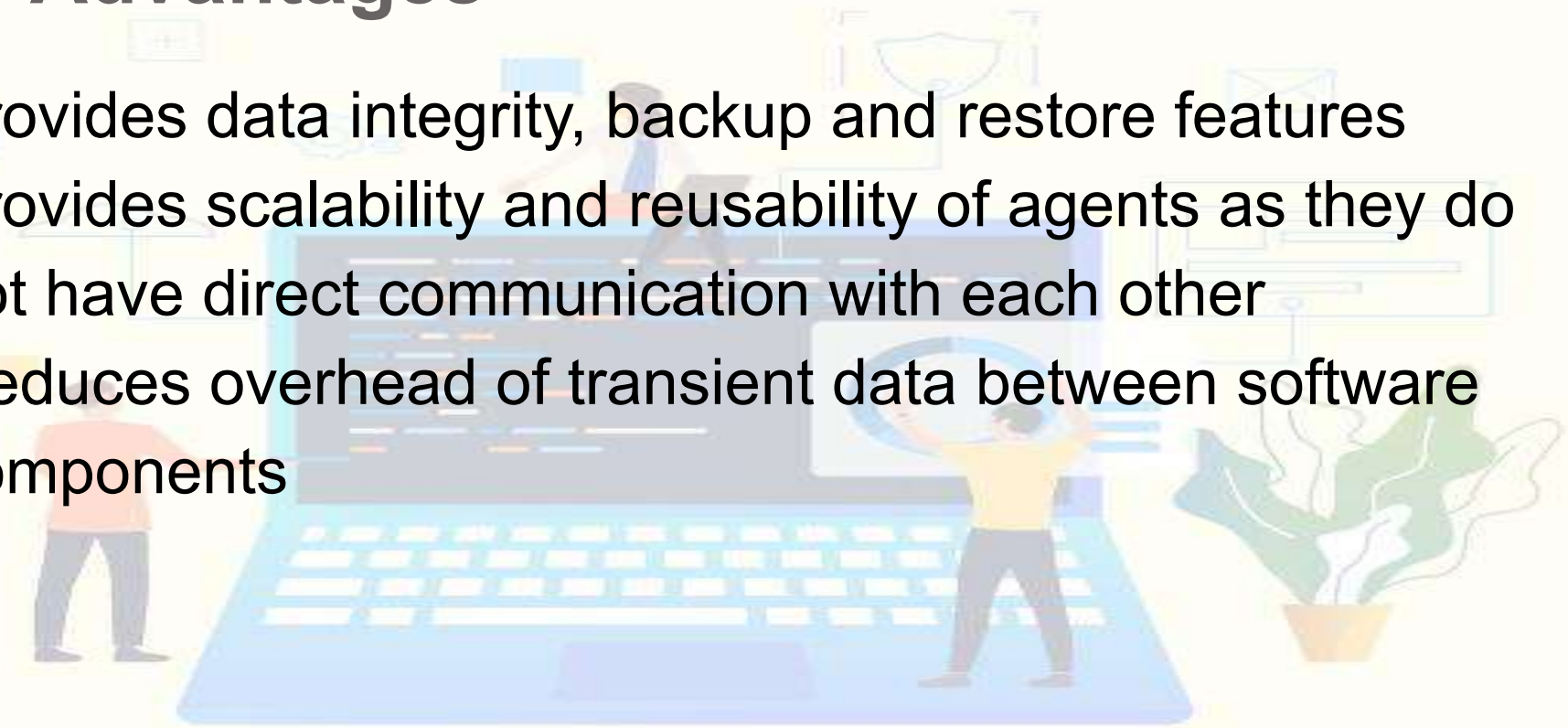
- *The client sends a request to the system to perform actions (e.g. insert data)*
- *The computational processes are independent and triggered by incoming requests*
- *If the types of transactions in an input stream of transactions trigger selection of processes to execute, then it is traditional database or repository architecture, or passive repository*
- *This approach is widely used in DBMS, library information system, the interface repository in CORBA, compilers and CASE (computer aided software engineering) environments*

Repository Architecture Style(RAS)



RAS Advantages

- Provides data integrity, backup and restore features
- Provides scalability and reusability of agents as they do not have direct communication with each other
- Reduces overhead of transient data between software components



RAS Disadvantages

- It is more vulnerable to failure and data replication or duplication is possible
- High dependency between data structure of data store and its agents
- Changes in data structure highly affect the clients
- Evolution of data is difficult and expensive
- Cost of moving data on network for distributed data

DCA Category: Blackboard Architecture Style

- The data store is active and its clients are passive therefore the logical flow is determined by the current data status in data store
- Has a blackboard component, acting as a central data repository, and an internal representation is built and acted upon by different computational elements

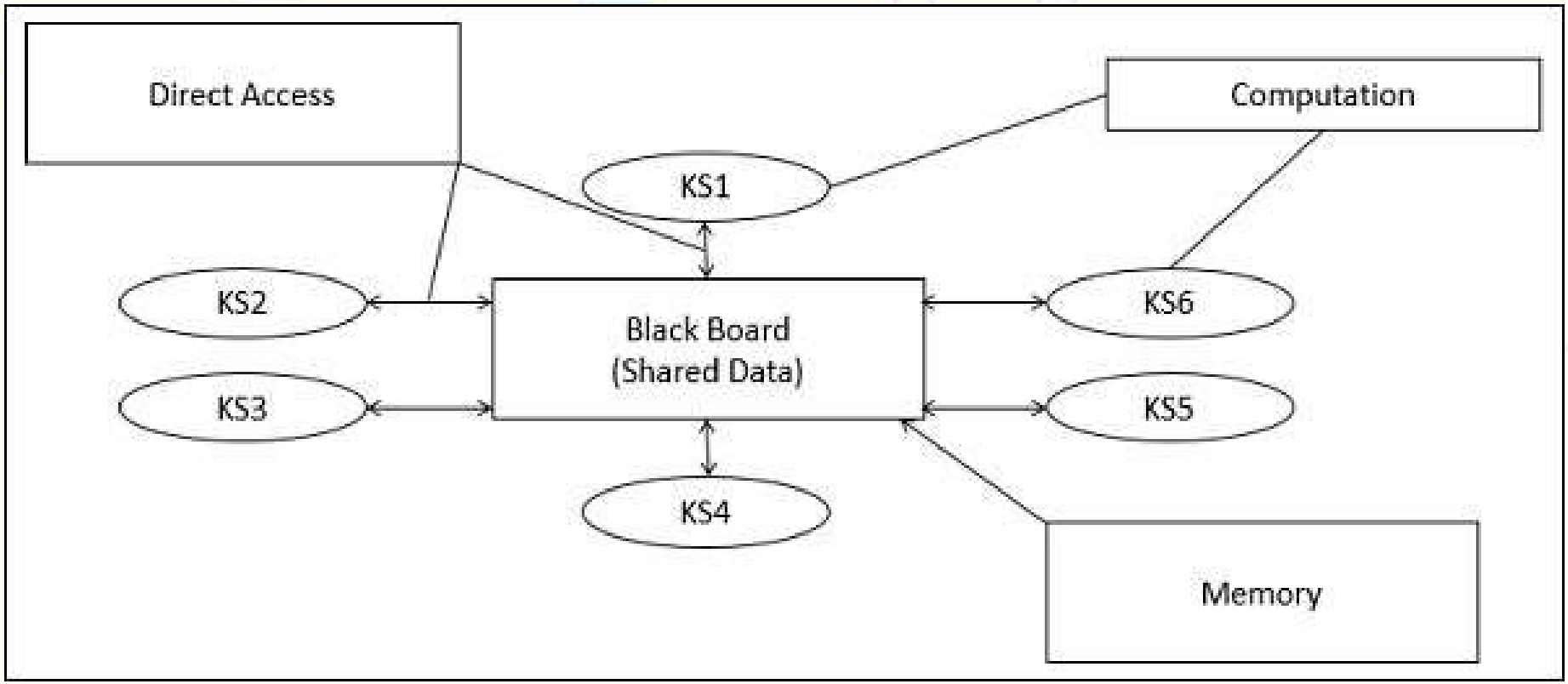
About Blackboard Architecture Style (BAS)

- *A number of components that act independently on the common data structure are stored in the blackboard*
- *In this style, the components interact only through the blackboard. The data-store alerts the clients whenever there is a data-store change*
- *The current state of the solution is stored in the blackboard and processing is triggered by the state of the blackboard*
- *The system sends notifications known as trigger and data to the clients when changes occur in the data*

About Blackboard Architecture Style (BAS)

- *This approach is found in certain AI applications and complex applications, such as speech recognition, image recognition, security system, and business resource management systems etc*
- *If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard and this shared data source is an active agent*
- *A major difference with traditional database systems is that the invocation of computational elements in a blackboard architecture is triggered by the current state of the blackboard, and not by external inputs*

Blackboard Architecture Style (BAS)



Parts of Blackboard Model

- **Knowledge Sources (KS)**
  - Also known as Listeners or Subscribers are distinct and independent units
  - They solve parts of a problem and aggregate partial results
  - Interaction among knowledge sources takes place uniquely through the blackboard

Parts of Blackboard Model

- **Blackboard Data Structure**
  - The problem-solving state data is organized into an application-dependent hierarchy
  - Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem
- **Control**
  - Control manages tasks and checks the work state

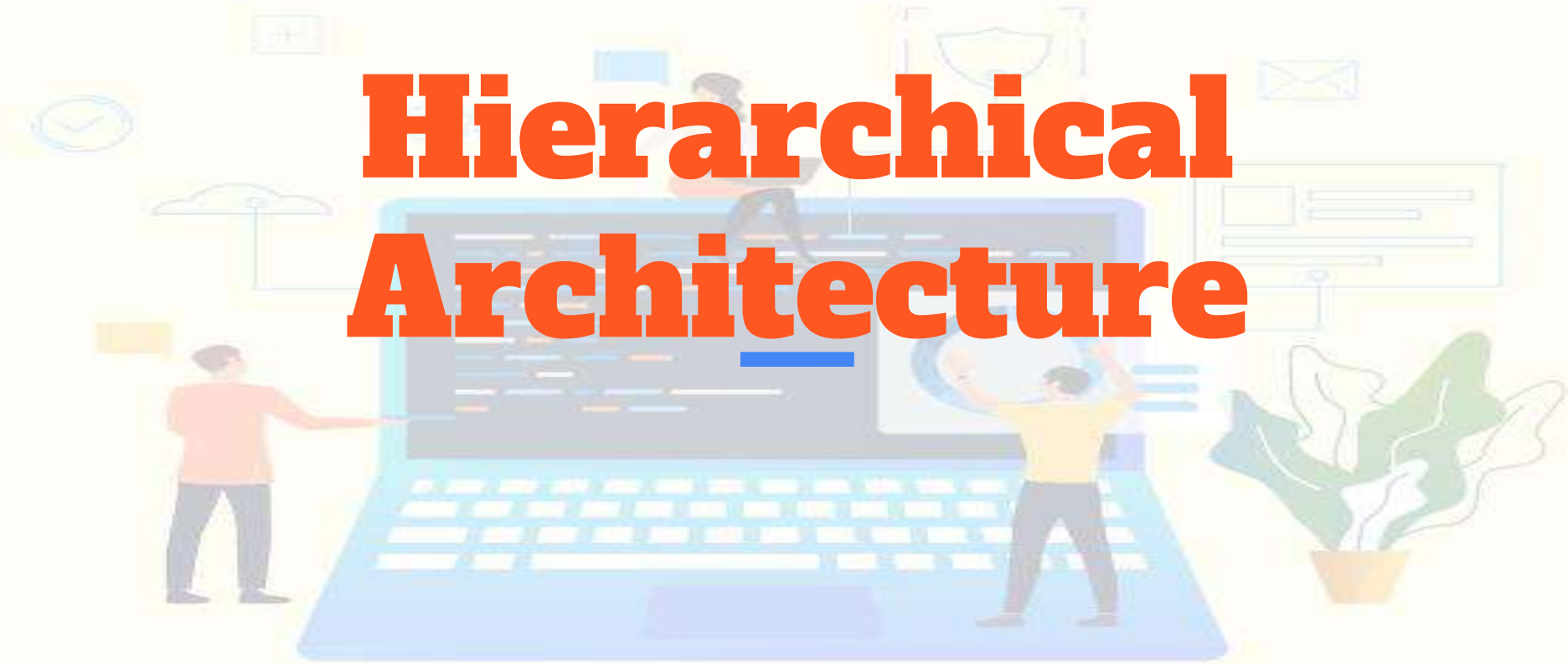
BAS Advantages

- Provides scalability which provides easy to add or update knowledge source
- Provides concurrency that allows all knowledge sources to work in parallel as they are independent of each other
- Supports experimentation for hypotheses
- Supports reusability of knowledge source agents

BAS Disadvantages

- The structure change of blackboard may have a significant impact on all of its agents as close dependency exists between blackboard and knowledge source
- It can be difficult to decide when to terminate the reasoning as only approximate solution is expected
- Problems in synchronization of multiple agents
- Major challenges in designing and testing of system

Hierarchical Architecture



Hierarchical Architecture

- Views the whole system as a hierarchy structure, in which the software system is decomposed into logical modules or subsystems at different levels in the hierarchy.
- This approach is typically used in designing system software such as network protocols & operating systems
- It is also used in organization of the class libraries such as .NET class library in namespace hierarchy
- All the design types can implement this hierarchical architecture & often combine with other architecture styles

System Software Hierarchy Design

- ➔ A low-level subsystem gives services to its adjacent upper level subsystems, which invoke the methods in the lower level
- ➔ The lower layer provides more specific functionality such as I/O services, transaction, scheduling, security services, etc
- ➔ The middle layer provides more domain dependent functions such as business logic and core processing services
- ➔ The upper layer provides more abstract functionality in the form of user interface such as GUIs, shell programming facilities, etc

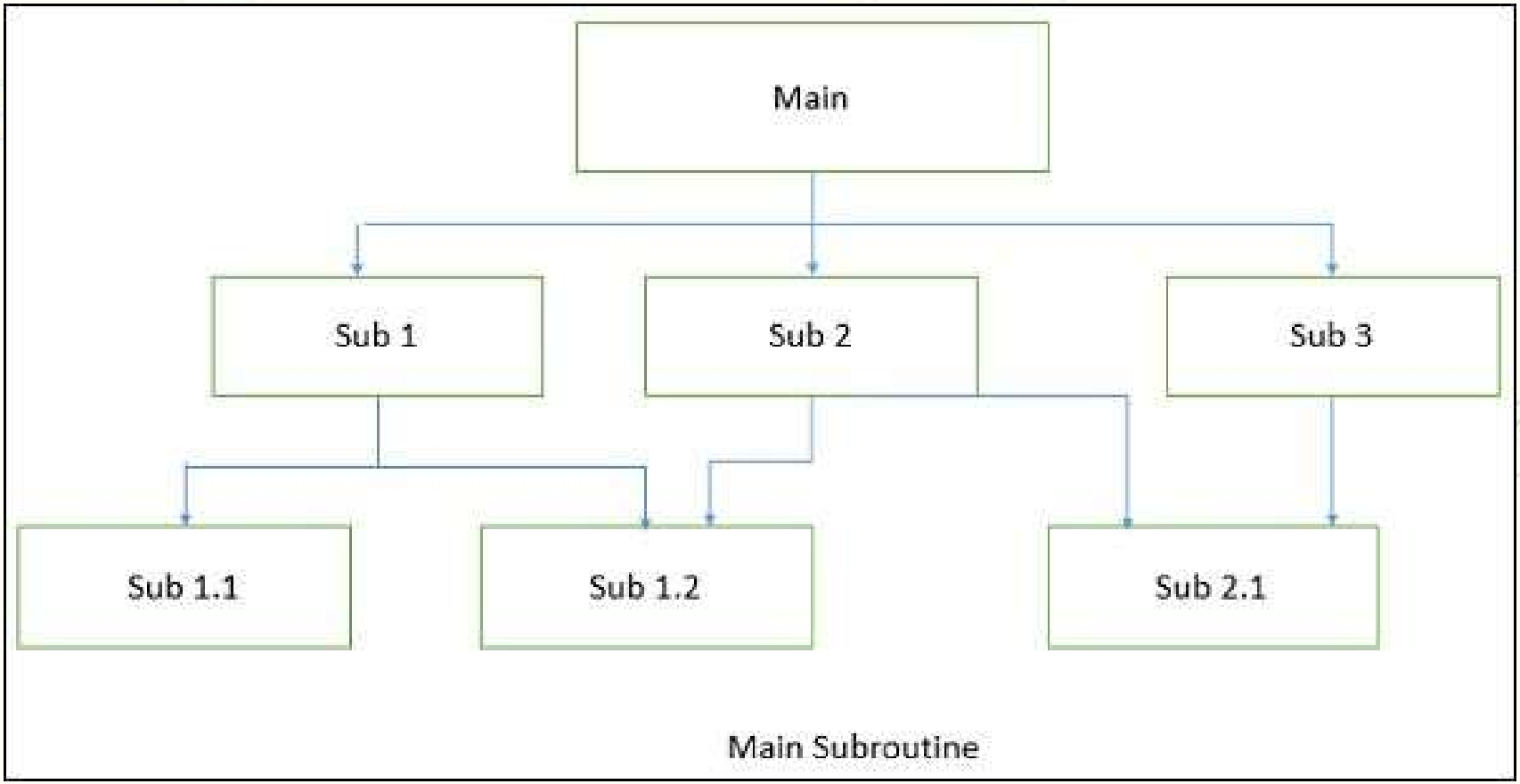
Main-Subroutine (Hierarchical Architectural Style)

- The aim of this style is to reuse the modules and freely develop individual modules or subroutine. In this style, a software system is divided into subroutines by using top-down refinement according to desired functionality of the system
  - These refinements lead vertically until the decomposed modules is simple enough to have its exclusive independent responsibility
- Functionality may be reused and shared by multiple callers in the upper layers

Main-Subroutine

- *There are two ways by which data is passed as parameters to subroutines:*
  - **Pass by Value** − Subroutines only use the past data, but can’t modify it
  - **Pass by Reference** − Subroutines use as well as change the value of the data referenced by the parameter

Main-Subroutine



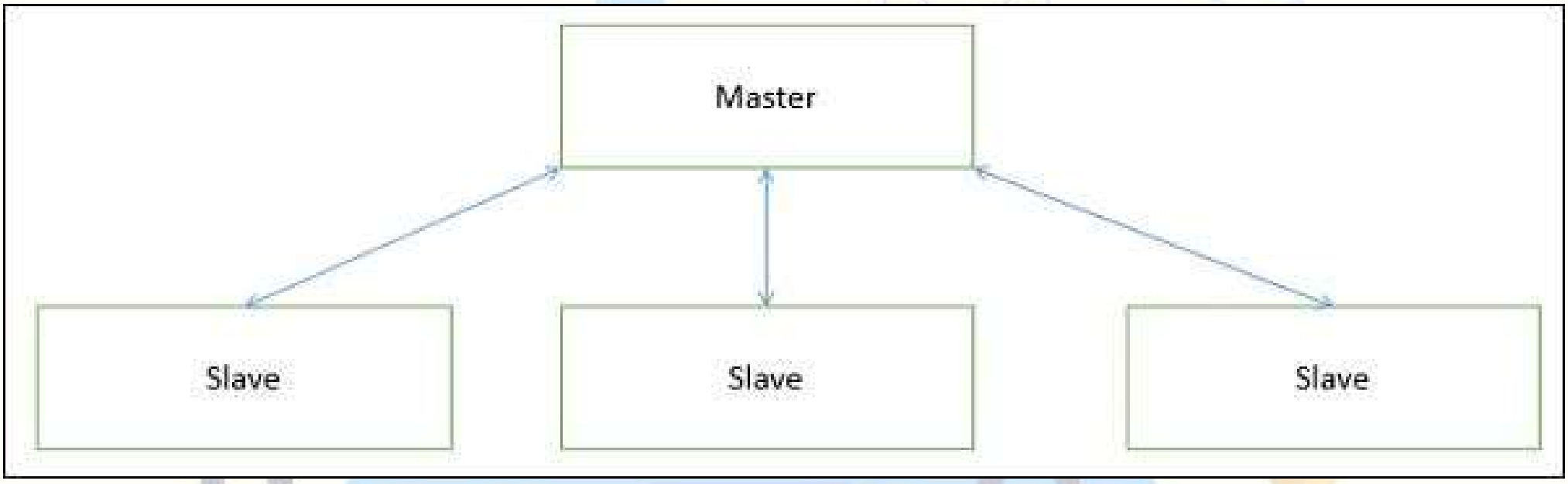
Main-Subroutine

- **Advantages:**
  - Easy to decompose the system based on hierarchy refinement
  - Can be used in a subsystem of object oriented design
- **Disadvantages:**
  - Vulnerable as it contains globally shared data.
  - Tight coupling may cause more ripple effects of changes

Master-Slave (Hierarchical Architectural Style)

- Applies the 'divide and conquer' principle and supports fault computation and computational accuracy
- A modification of the main-subroutine architecture that provides reliability of system and fault tolerance
- In this architecture:
  - Slaves provide duplicate services to the master, and the master chooses a particular result among slaves by a certain selection strategy
  - The slaves may perform the same functional task by different algorithms and methods or totally different functionality. It includes parallel computing in which all the slaves can be executed in parallel

Master-Slave

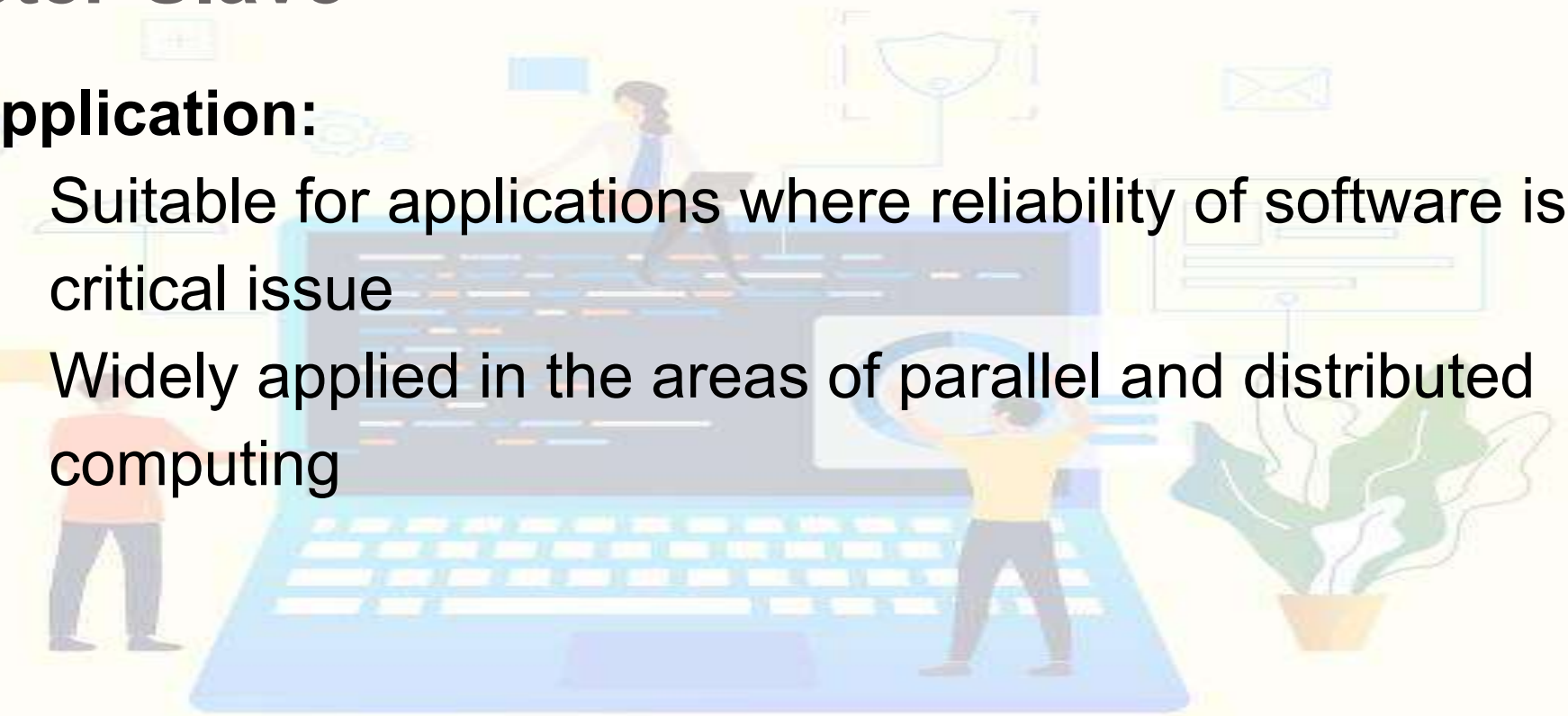


Master-Slave 5 Steps Implementation

- Specify how the computation of the task can be divided into a set of equal sub-tasks and identify the sub-services that are needed to process a sub-task
- Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks
- Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks
- Implement the slave components according to the specifications developed in the previous step
- Implement the master according to the specifications developed in step 1 to 3

Master-Slave

- **Application:**
  - Suitable for applications where reliability of software is critical issue
  - Widely applied in the areas of parallel and distributed computing

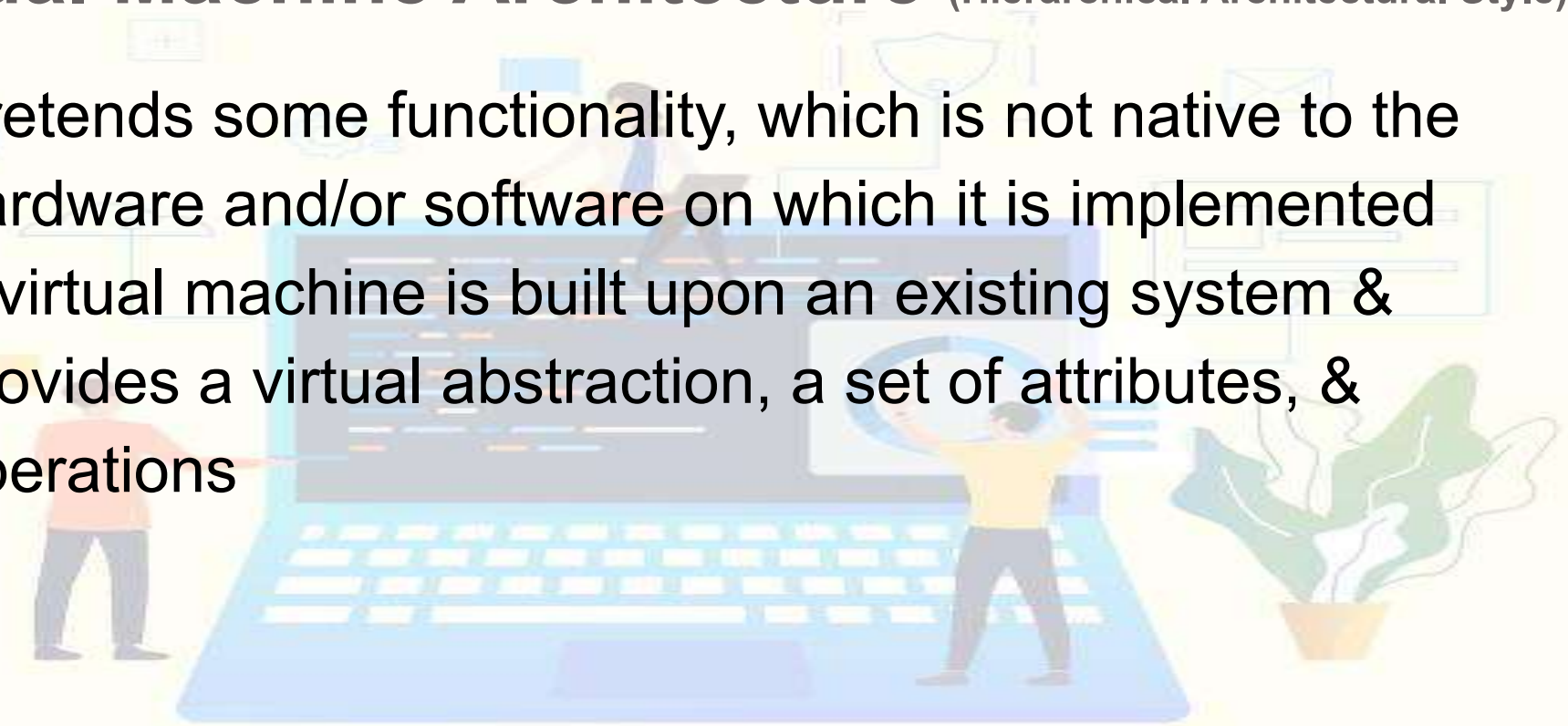


Master-Slave

- **Advantages:**
  - Faster computation and easy scalability
  - Provides robustness as slaves can be duplicated
  - Slave can be implemented differently to minimize semantic errors.
- **Disadvantages:**
  - Communication overhead.
  - Not all problems can be divided.
  - Hard to implement and portability issue.

Virtual Machine Architecture (Hierarchical Architectural Style)

- Pretends some functionality, which is not native to the hardware and/or software on which it is implemented
- A virtual machine is built upon an existing system & provides a virtual abstraction, a set of attributes, & operations



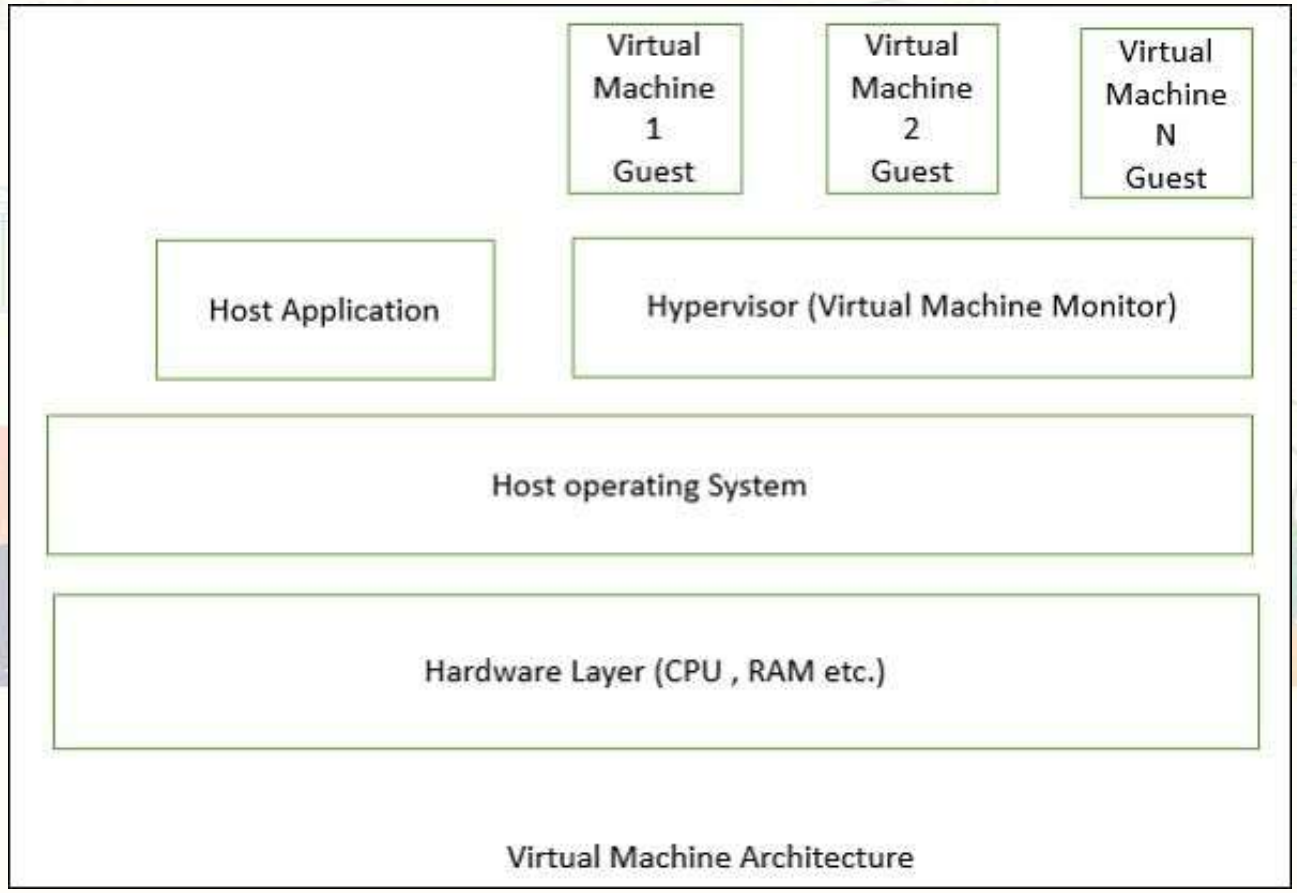
Virtual Machine Architecture (VMA)

- In virtual machine architecture,
  - The master uses the ‘same’ subservice’ from the slave and performs functions such as split work, call slaves, and combine results
  - It allows developers to simulate and test platforms, which have not yet been built, and simulate "disaster" modes that would be too complex, costly, or dangerous to test with the real system
- In most cases, a virtual machine splits a programming language or application environment from an execution platform
- The main objective is to provide ***portability***

Virtual Machine Architecture (VMA)

- Interpretation of a particular module via a Virtual Machine may be perceived as −
  - The interpretation engine chooses an instruction from the module being interpreted
  - Based on the instruction, the engine updates the virtual machine’s internal state and the above process is repeated

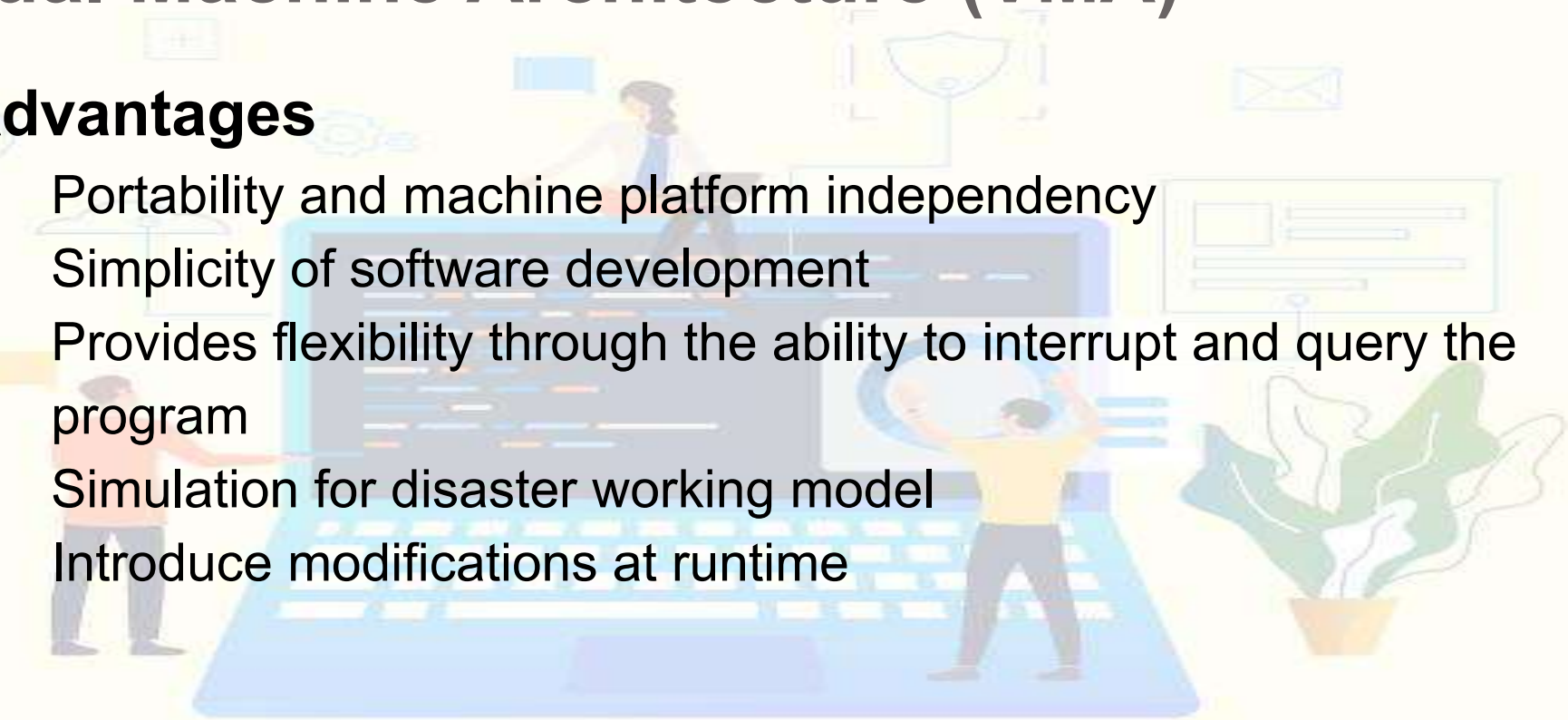
Virtual Machine Architecture (VMA)



Virtual Machine Architecture (VMA)

- **Applications**
  - Suitable for solving a problem by simulation or translation if there is no direct solution
  - Sample applications include interpreters of microprogramming, XML processing, script command language execution, rule-based system execution, Smalltalk and Java interpreter typed programming language
  - Common examples of virtual machines are interpreters, rule-based systems, syntactic shells, and command language processors

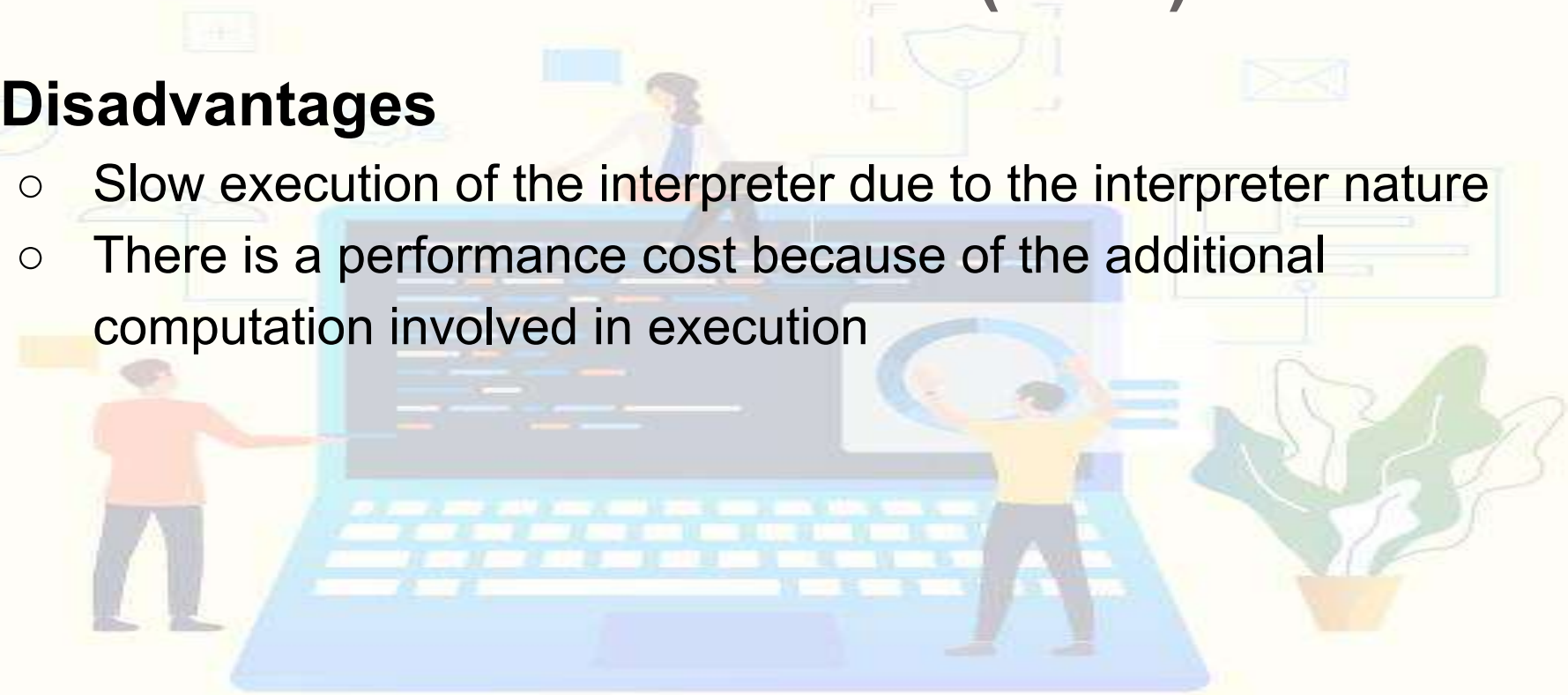
Virtual Machine Architecture (VMA)



- **Advantages**
  - Portability and machine platform independency
  - Simplicity of software development
  - Provides flexibility through the ability to interrupt and query the program
  - Simulation for disaster working model
  - Introduce modifications at runtime

Virtual Machine Architecture (VMA)

- **Disadvantages**
  - Slow execution of the interpreter due to the interpreter nature
  - There is a performance cost because of the additional computation involved in execution



Layered Style (Hierarchical Architectural Style)

- In this approach, the system is decomposed into a number of higher and lower layers in a hierarchy, and each layer has its own sole responsibility in the system.
  - Each layer consists of a group of related classes that are encapsulated in a package, in a deployed component, or as a group of subroutines in the format of method library or header file
  - Each layer provides service to the layer above it and serves as a client to the layer below i.e. request to layer i +1 invokes the services provided by the layer i via the interface of layer i
  - The response may go back to the layer i +1 if the task is completed; otherwise layer i continually invokes services from layer i -1 below

Layered Style

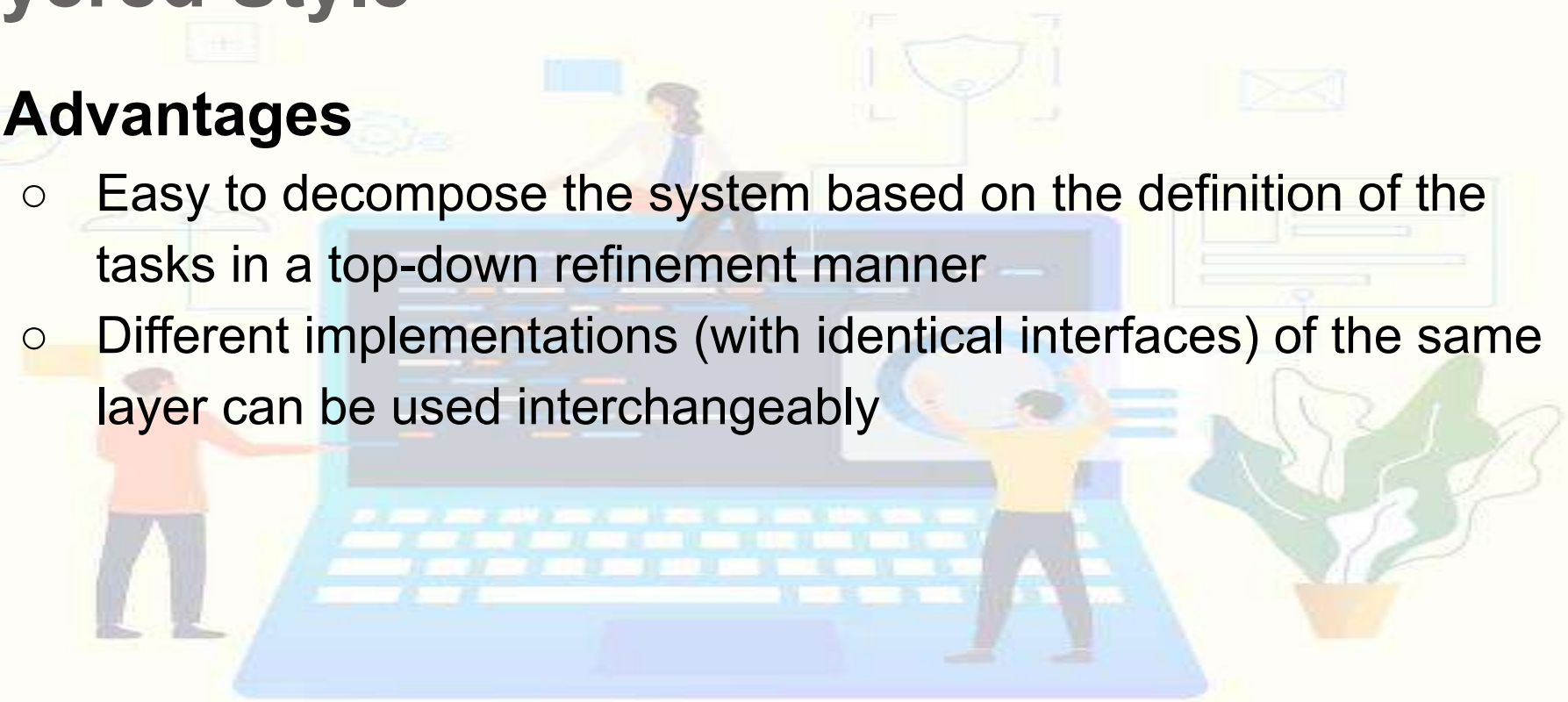
- **Applications**
  - Applications that involve distinct classes of services that can be organized hierarchically
  - Any application that can be decomposed into application-specific and platform-specific portions
  - Applications that have clear divisions between core services, critical services, and user interface services, etc

Layered Style

- **Advantages**
  - Design based on incremental levels of abstraction
  - Provides enhancement independence as changes to the function of one layer affects at most two other layers
  - Separation of the standard interface and its implementation
  - Implemented by using component-based technology which makes the system much easier to allow for plug-and-play of new components

Layered Style

- **Advantages**
  - Easy to decompose the system based on the definition of the tasks in a top-down refinement manner
  - Different implementations (with identical interfaces) of the same layer can be used interchangeably

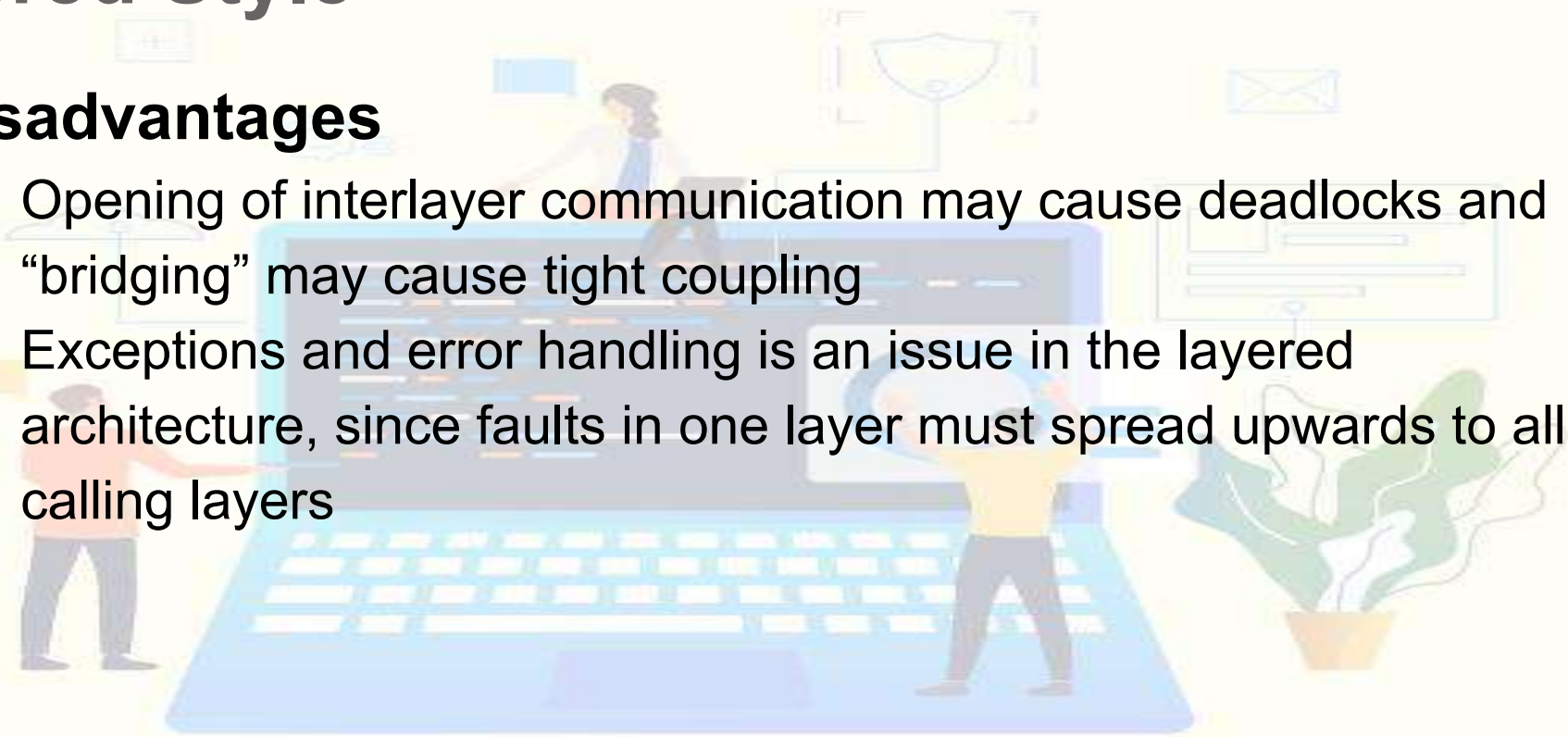


Layered Style

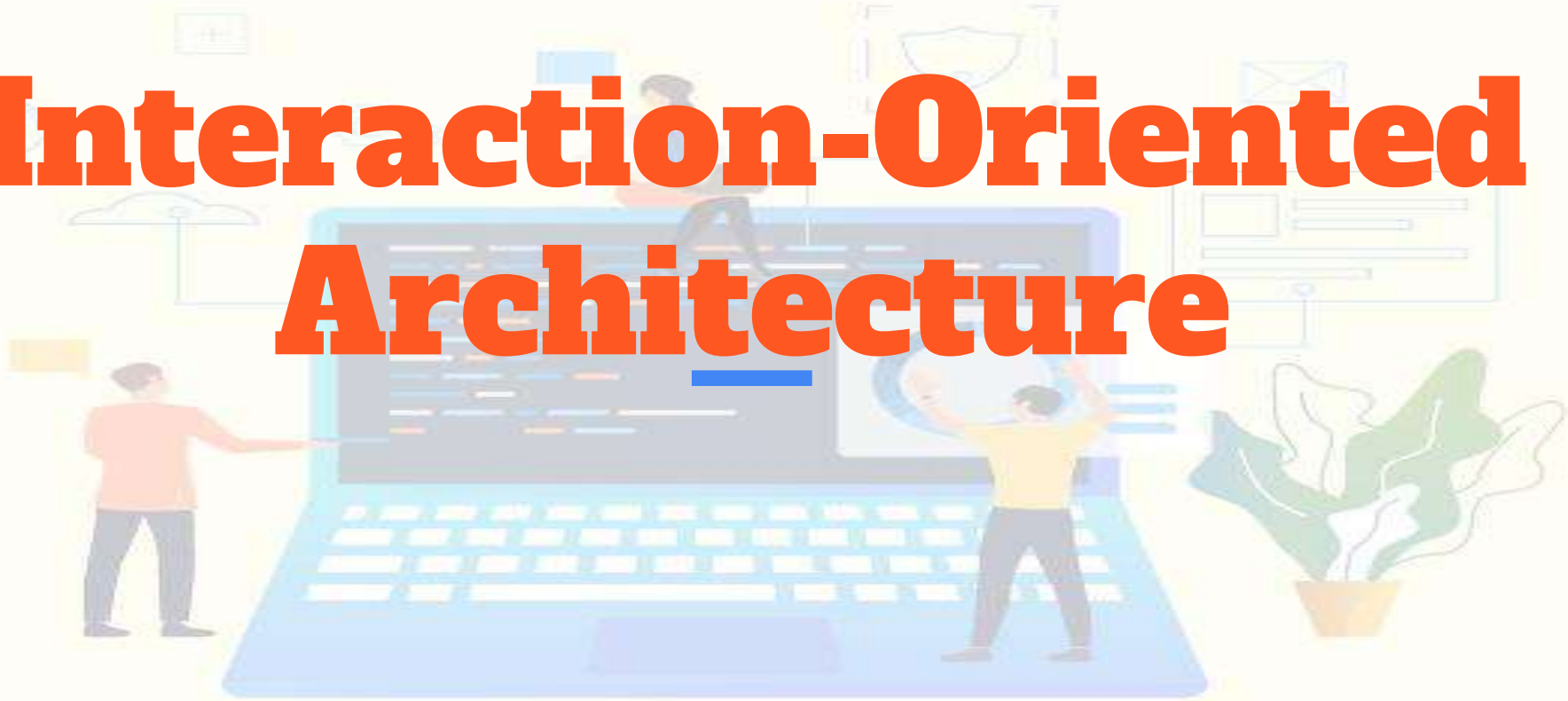
- **Disadvantages**
  - Many applications or systems are not easily structured in a layered fashion
  - Lower runtime performance since a client's request or a response to client must go through potentially several layers
  - There are also performance concerns on overhead on the data marshaling and buffering by each layer

Layered Style

- **Disadvantages**
  - Opening of interlayer communication may cause deadlocks and “bridging” may cause tight coupling
  - Exceptions and error handling is an issue in the layered architecture, since faults in one layer must spread upwards to all calling layers



Interaction-Oriented Architecture



Interaction-Oriented Architecture (IOA)

- ***Primary objective:*** To separate the interaction of user from data abstraction and business data processing
- ***Three major partitions:*** Data module, Control module, & View presentation module
- ***Two major styles:*** Model-View-Controller (MVC) & Presentation-Abstraction-Control (PAC)

IOA Major Partitions

- **Data module** − provides the data abstraction and all business logic
- **Control module** − identifies the flow of control and system configuration actions
- **View presentation module** − responsible for visual or audio presentation of data output & also provides an interface for user input

Model-View-Controller (MVC)

- Decomposes a given software application into three interconnected parts that help in separating the internal representations of information from the information presented to or accepted from the user
- ***Modules:***
  - **Model** – encapsulation the underlying data and business logic
  - **Controller** – respond to user action & direct the application flow
  - **View –** formats & present the data from model to user

MVC: Model

- A central component of MVC that directly manages the data, logic, and constraints of an application
- Consists of ***data components***, which maintain the raw application data & application logic for interface:
  - *It is an independent user interface & captures the behavior of application problem domain*
  - *It is the domain-specific software simulation or implementation of the application's central structure*
  - *When there has been change in its state, it gives notification to its associated view to produce updated output and the controller to change the available set of commands*

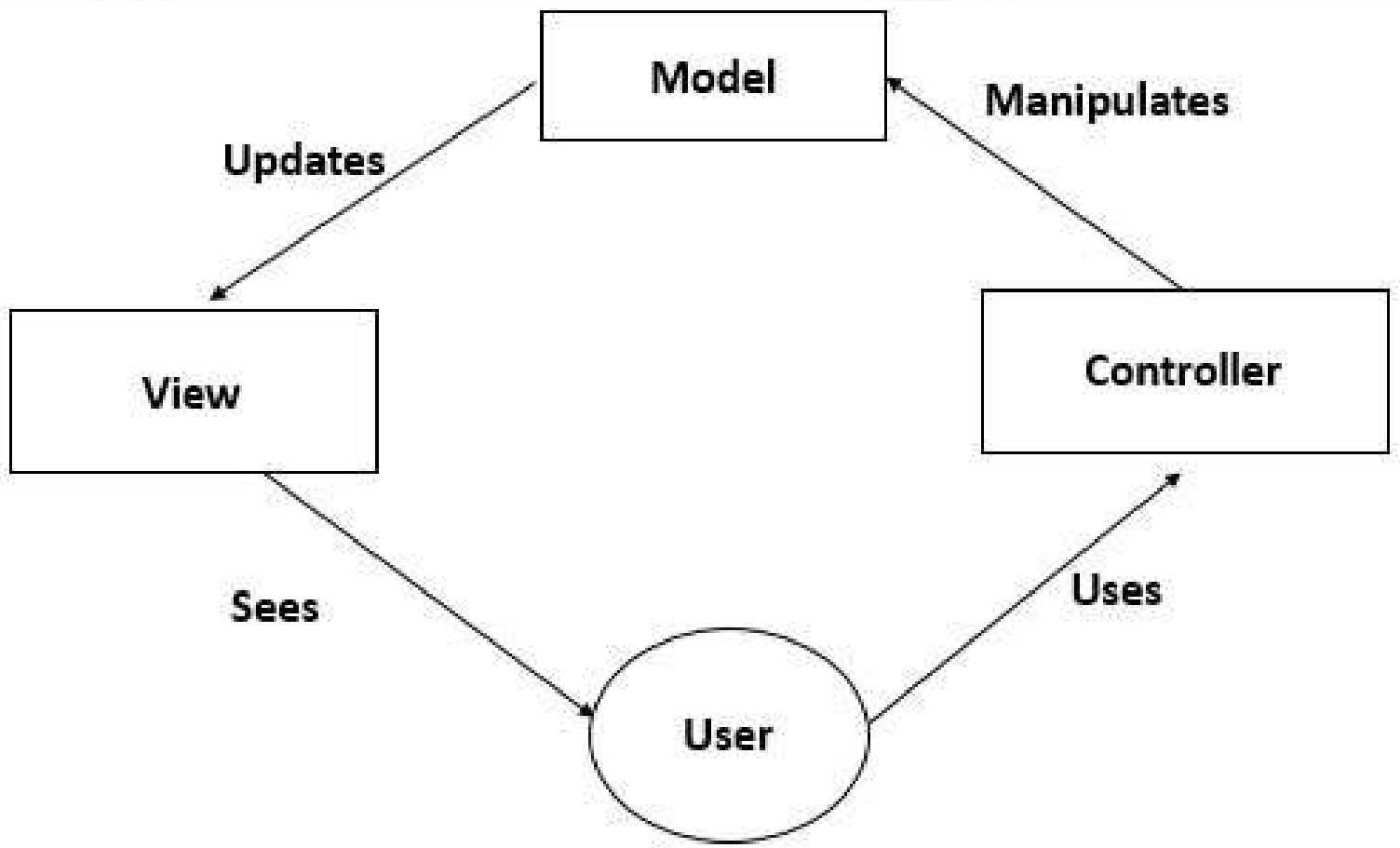
MVC: View

- Can be used to represent any output of information in graphical form such as diagram or chart.
- Consists of presentation components which provide the visual representations of data:
  - *Views request information from their model and generate an output representation to the user*
  - *Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.*

MVC: Controller

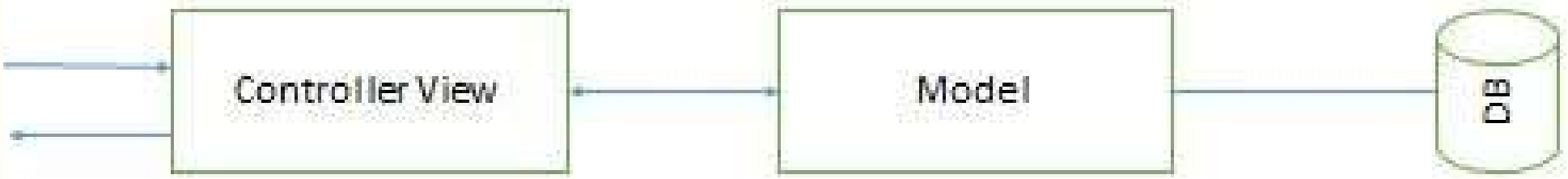
- Accepts an input and converts it to commands for the model or view
- Consists of input processing components which handle input from the user by modifying the model:
  - *It acts as an interface between the associated models and views and the input devices.*
  - *It can send commands to the model to update the model's state and to its associated view to change the view's presentation of the model*

Typical Collaboration of MVC Component



MVC-I Architecture

- A simple version of MVC architecture where the system
- ***Divided into 2 subsystems:***
  - **Controller-View** − acts as input /output interface and processing is done
  - **Model** − provides all the data and domain services



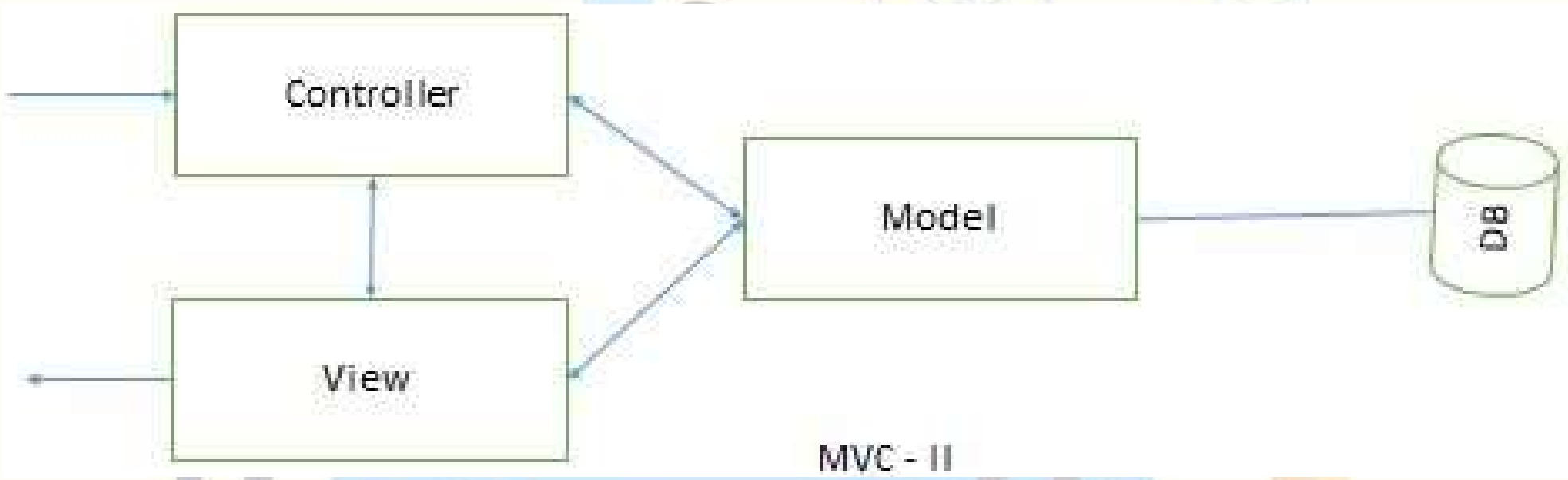
MVC-I Architecture

- The connection between controller-view and model can be designed in a pattern of subscribe-notify whereby the controller-view subscribes to model and model notifies controller-view of any changes
  - *The model module notifies controller-view module of any data changes so that any graphics data display will be changed accordingly*
  - *The controller also takes appropriate action upon the changes*

MVC-II Architecture

- An enhancement of MVC-I architecture in which the view module and the controller module are separate.
  - *The model module plays an active role as in MVC-I by providing all the core functionality and data supported by database*
  - *The view module presents data while controller module accepts input request, validates input data, initiates the model, the view, their connection, and also dispatches the task*

MVC-II Architecture



MVC Application

- Effective for interactive applications where multiple views are needed for a single data model and easy to plug-in a new or change interface view
- Suitable for applications where there are clear divisions between the modules so that different professionals can be assigned to work on different aspects of such applications concurrently

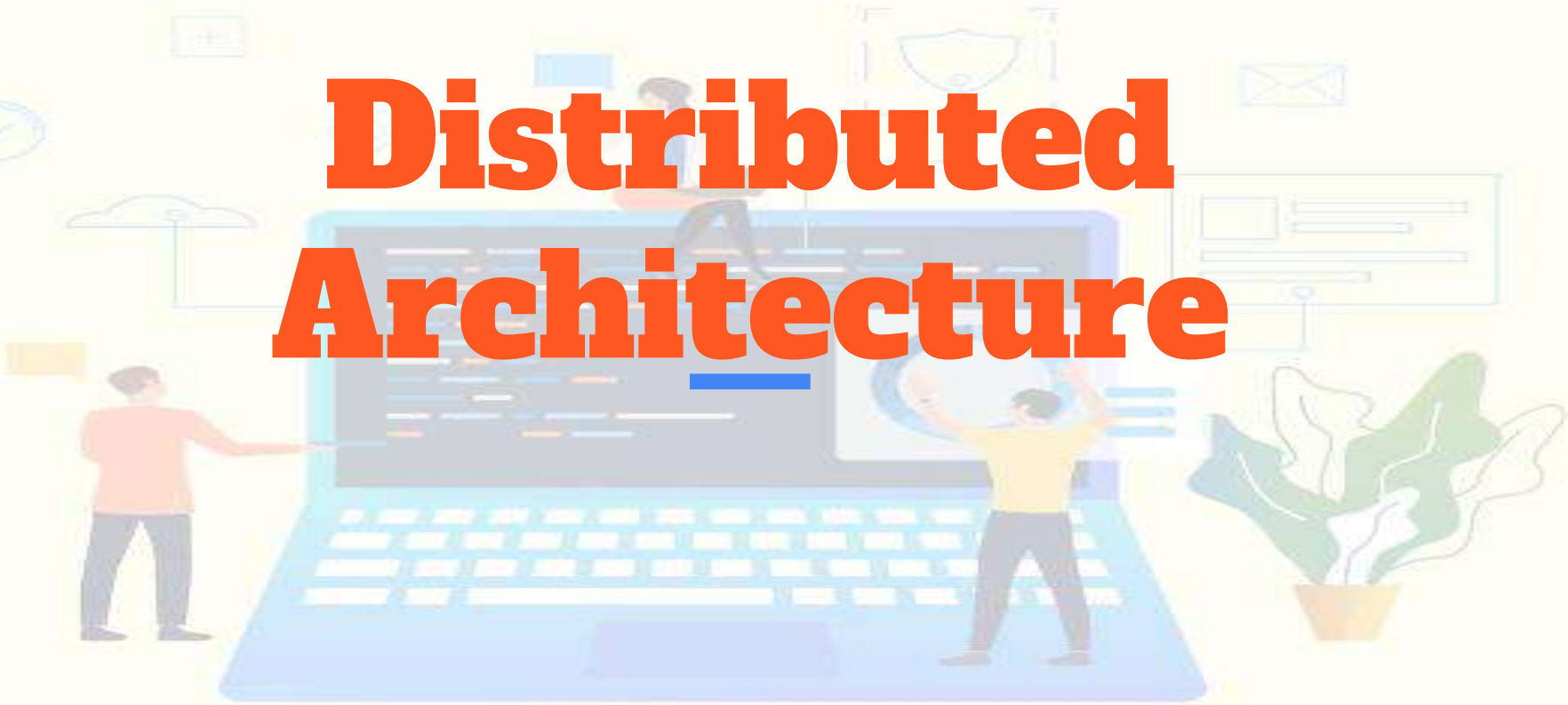
MVC Advantages

- There are many MVC vendor framework toolkits available
- Multiple views synchronized with same data model
- Easy to plug-in new or replace interface views
- Used for application development where graphics expertise professionals, programming professionals, and database development professionals are working in a designed project team

MVC Disadvantages

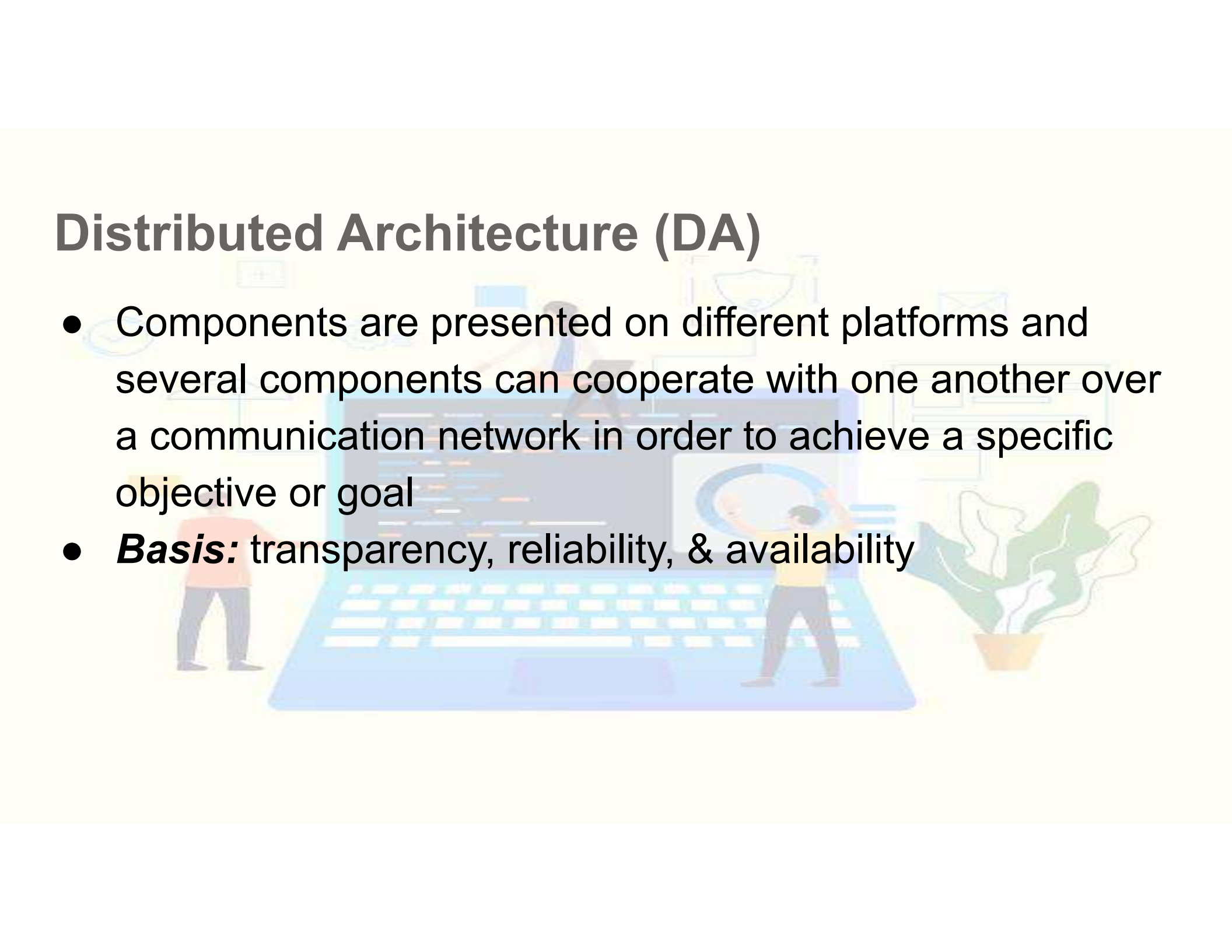
- Not suitable for agent-oriented applications such as interactive mobile and robotics applications
- Multiple pairs of controllers and views based on the same data model make any data model change expensive
- The division between the View and the Controller is not clear in some cases

Distributed Architecture



Distributed Architecture (DA)

- Components are presented on different platforms and several components can cooperate with one another over a communication network in order to achieve a specific objective or goal
- ***Basis:*** transparency, reliability, & availability



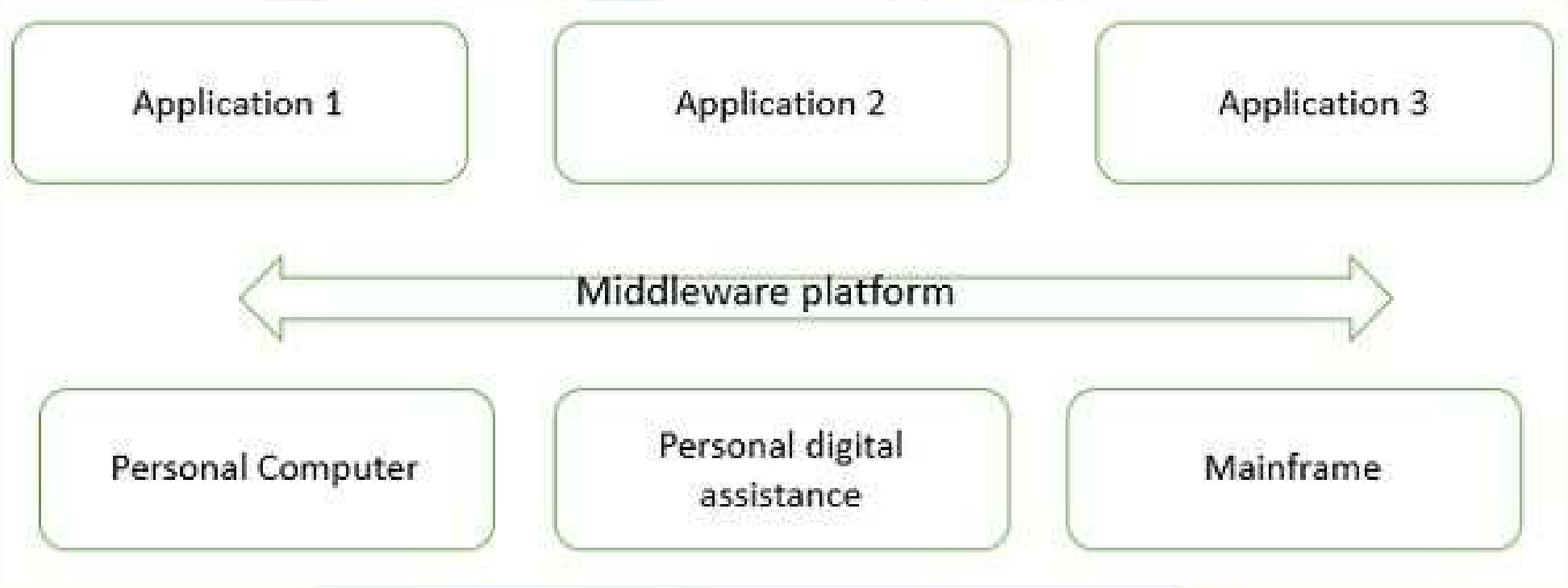
Distributed Architecture (DA)

- *In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers*
- *A distributed system can be demonstrated by the client-server architecture which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA)*
- *There are several technology frameworks to support distributed architectures, including .NET, J2EE, CORBA, .NET Web services, AXIS Java Web services, and Globus Grid services*

Distributed Architecture (DA)

- *Middleware is an infrastructure that appropriately supports the development and execution of distributed applications. It provides a buffer between the applications and the network*
- *It sits in the middle of system and manages or supports the different components of a distributed system. Examples are transaction processing monitors, data convertors and communication controllers etc*

Middleware as an Infrastructure



Different Forms of Transparency

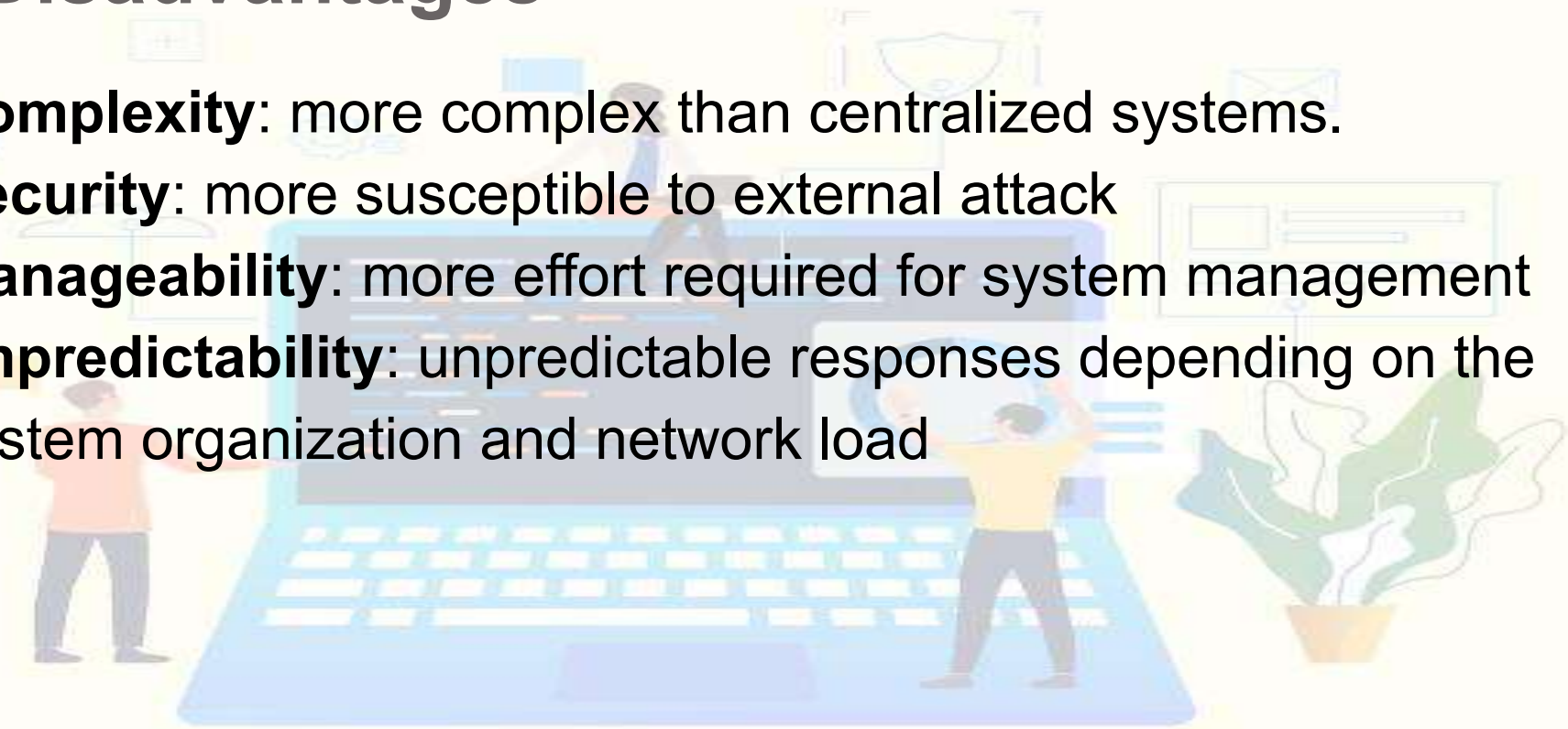
- **Access**: hides the way in which resources are accessed and the differences in data platform
- **Location**: hides where resources are located
- **Technology**: hides different technologies such as programming language and OS from user
- **Migration/Relocation**: hide resources that may be moved to another location which are in use
- **Replication**: hide resources that may be copied at several location
- **Concurrency**: hide resources that may be shared with other users
- **Failure**: hides failure and recovery of resources from user
- **Persistence**: hides whether a resource is in memory or disk

DA Advantages

- **Resource sharing**: sharing of hardware and software resources
- **Openness**: flexibility of using hardware and software of different vendors
- **Concurrency**: concurrent processing to enhance performance
- **Scalability**: increased throughput by adding new resources
- **Fault tolerance**: the ability to continue in operation after a fault has occurred

DA Disadvantages

- **Complexity**: more complex than centralized systems.
- **Security**: more susceptible to external attack
- **Manageability**: more effort required for system management
- **Unpredictability**: unpredictable responses depending on the system organization and network load



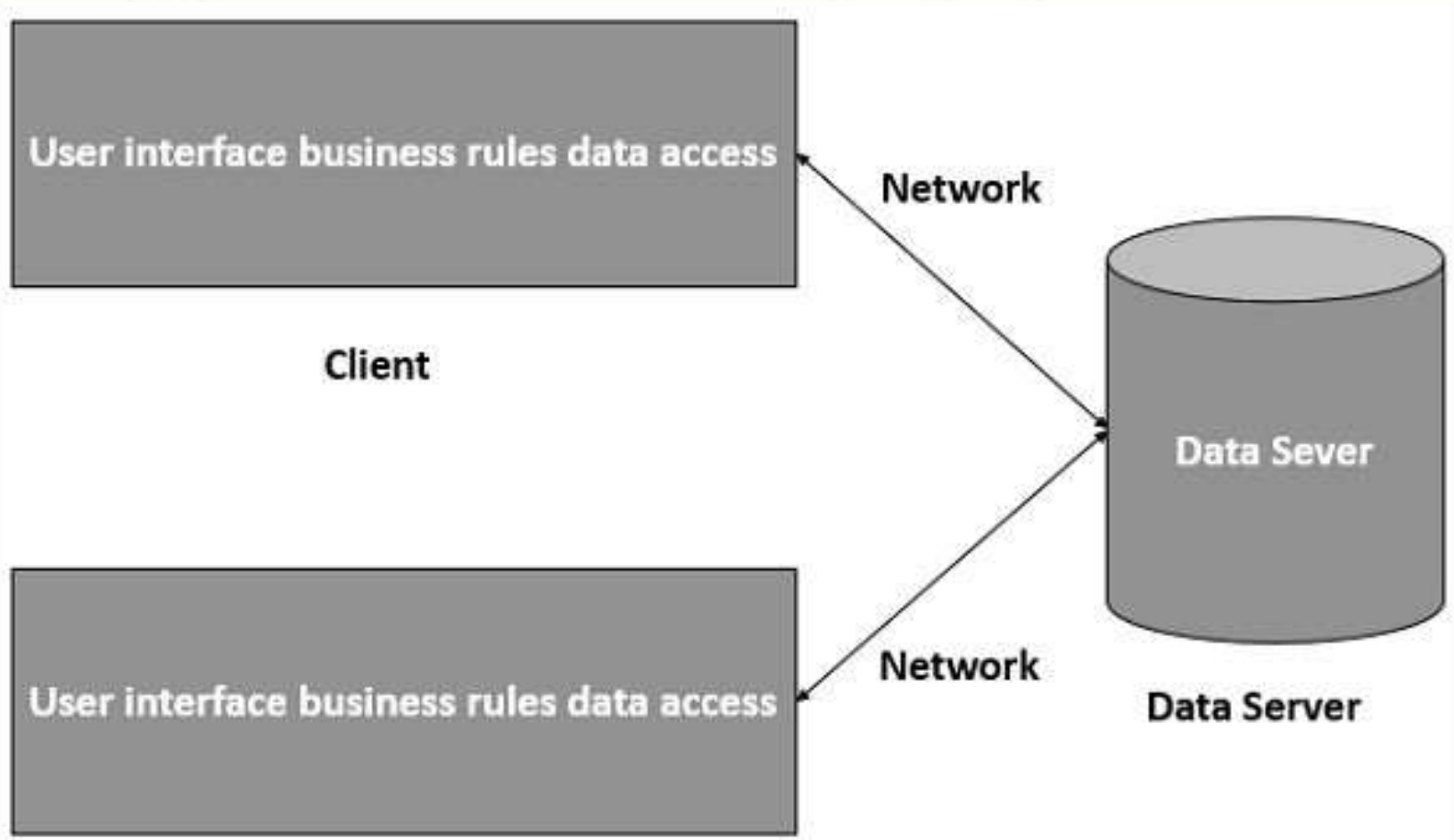
Client-Server Architecture (CSA)

- Most common distributed system architecture which decomposes the system into two major subsystems or logical processes:
  - **Client**: *the first process that issues a request to the second process i.e. the server*
  - **Server**: *the second process that receives the request, carries it out, and sends a reply to the client*

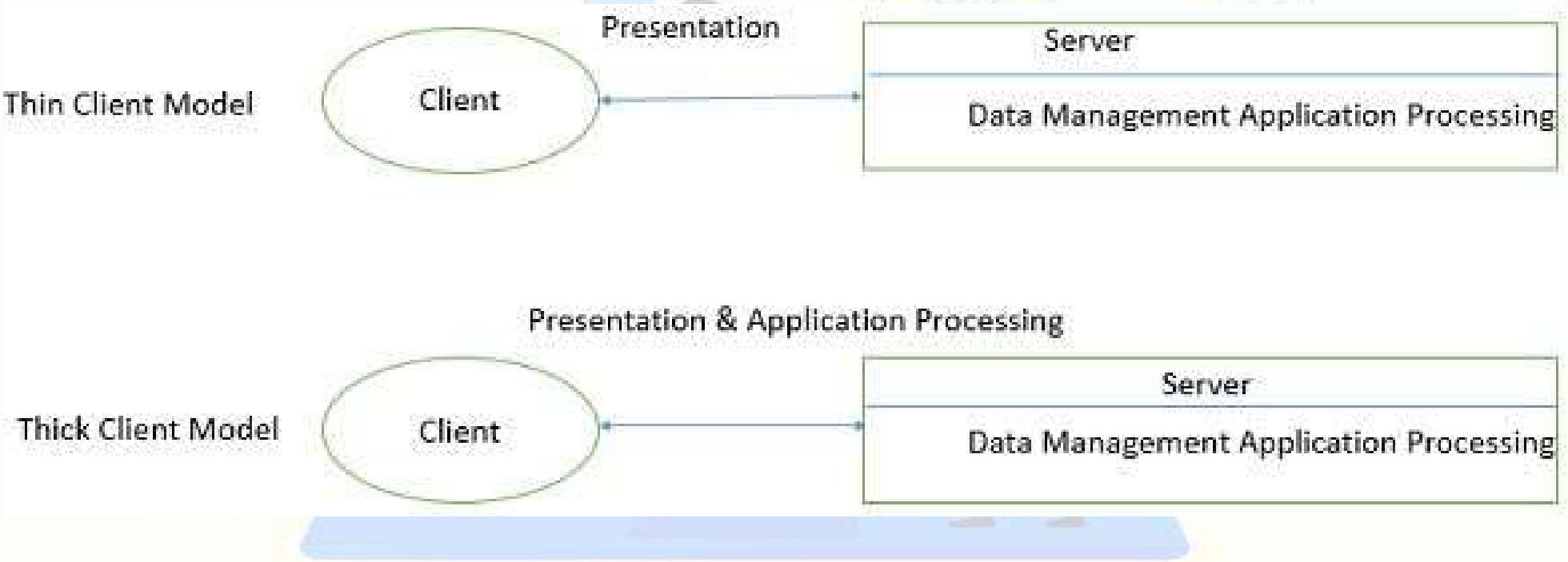
Client-Server Architecture CSA

- In this architecture, the application is modelled as a set of services that are provided by servers and a set of clients that use these services
- The servers need not know about clients, but the clients must know the identity of servers, & the mapping of processors to processes is not necessarily 1 : 1

2–Tier Client-Server Architecture



Models of Client-Server Architecture



Models of Client-Server Architecture

Thin-Client Model

- In thin-client model, all the application processing and data management is carried by the server
- The client is simply responsible for running the presentation software
  - Used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client
  - A major disadvantage is that it places a heavy processing load on both the server and the network

Models of Client-Server Architecture

Thick/Fat-Client Model

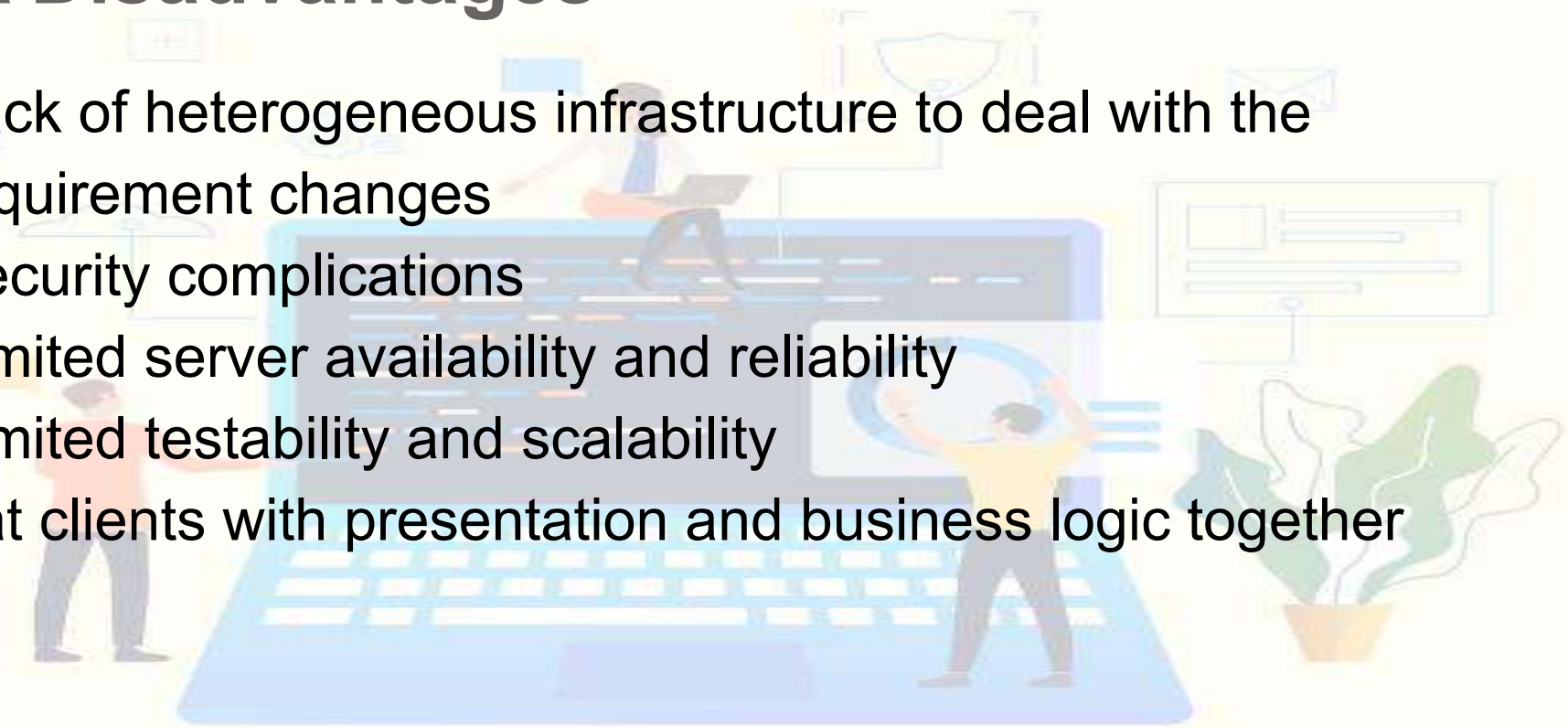
- In thick-client model, the server is only in charge for data management
- The software on the client implements the application logic and the interactions with the system user
  - Most appropriate for new C/S systems where the capabilities of the client system are known in advance
  - More complex than a thin client model especially for management
  - New versions of the application have to be installed on all clients

CSA Advantages

- Separation of responsibilities such as user interface presentation and business logic processing
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications
- It makes it easy to migrate or integrate existing applications into a distributed environment
- It also makes effective use of resources when a large number of clients are accessing a high-performance server

CSA Disadvantages

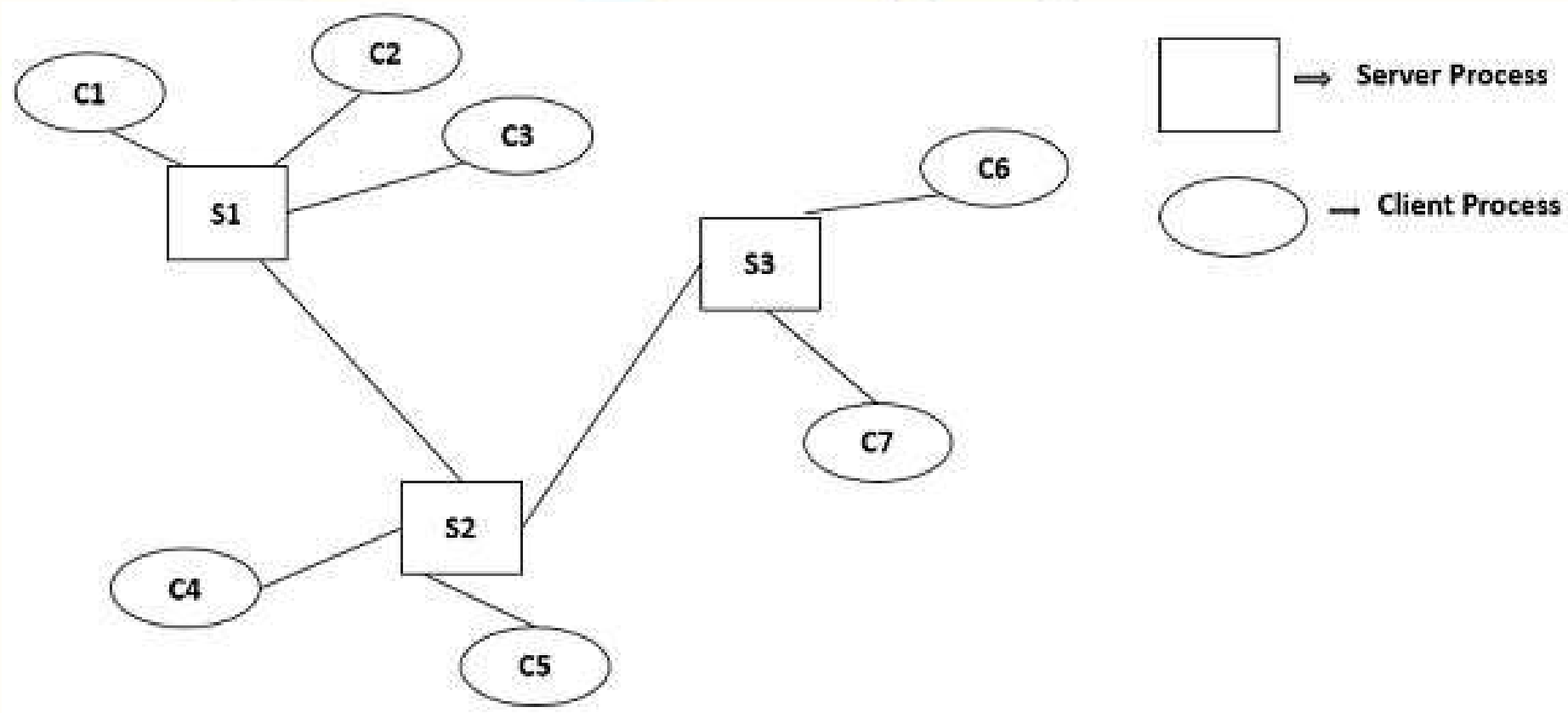
- Lack of heterogeneous infrastructure to deal with the requirement changes
- Security complications
- Limited server availability and reliability
- Limited testability and scalability
- Fat clients with presentation and business logic together



Multi-Tier Architecture (MTA)

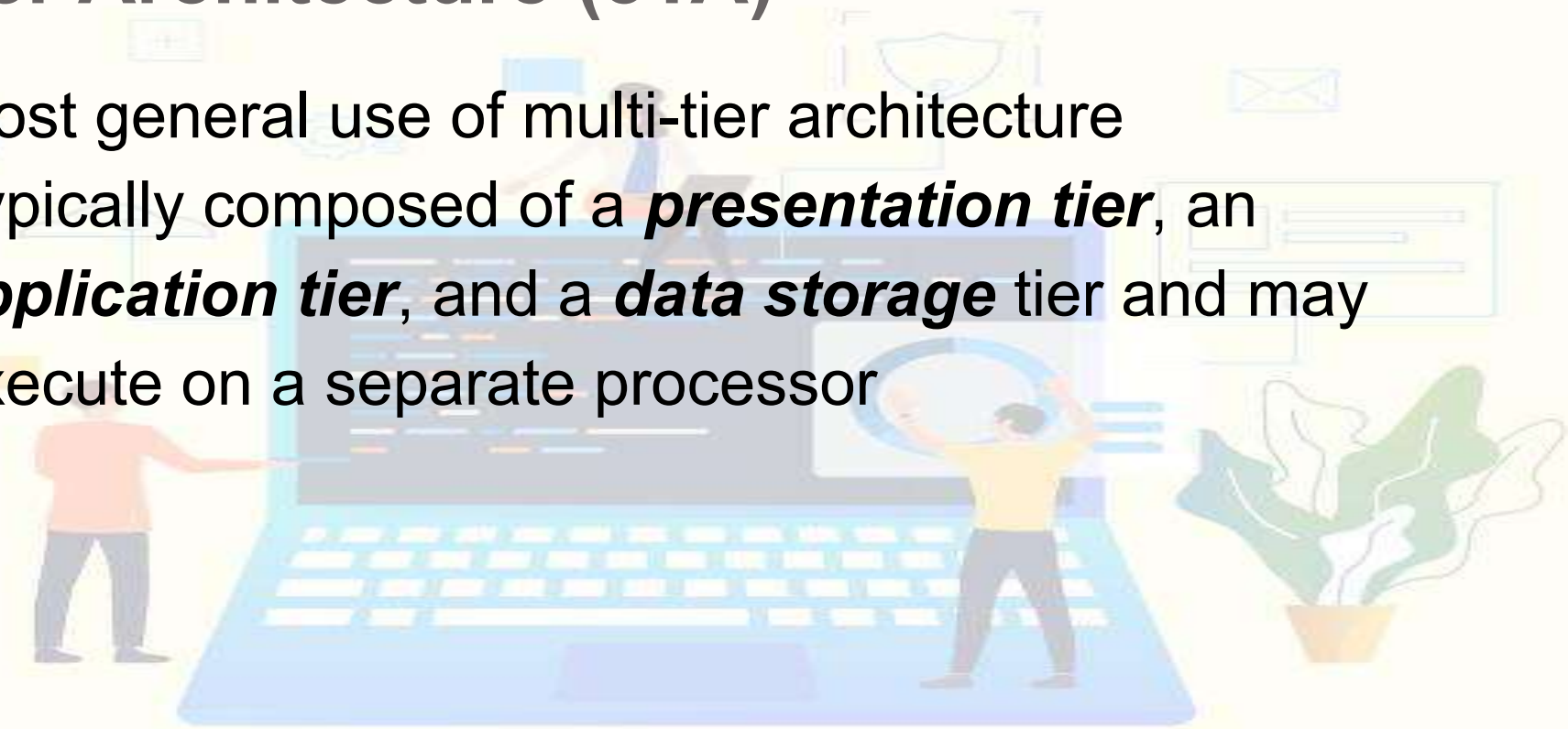
- **N-tier Architecture**
- A client–server architecture in which the functions such as presentation, application processing, and data management are physically separated
  - By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application
  - It provides a model by which developers can create flexible and reusable applications

Multi-Tier Architecture (MTA)

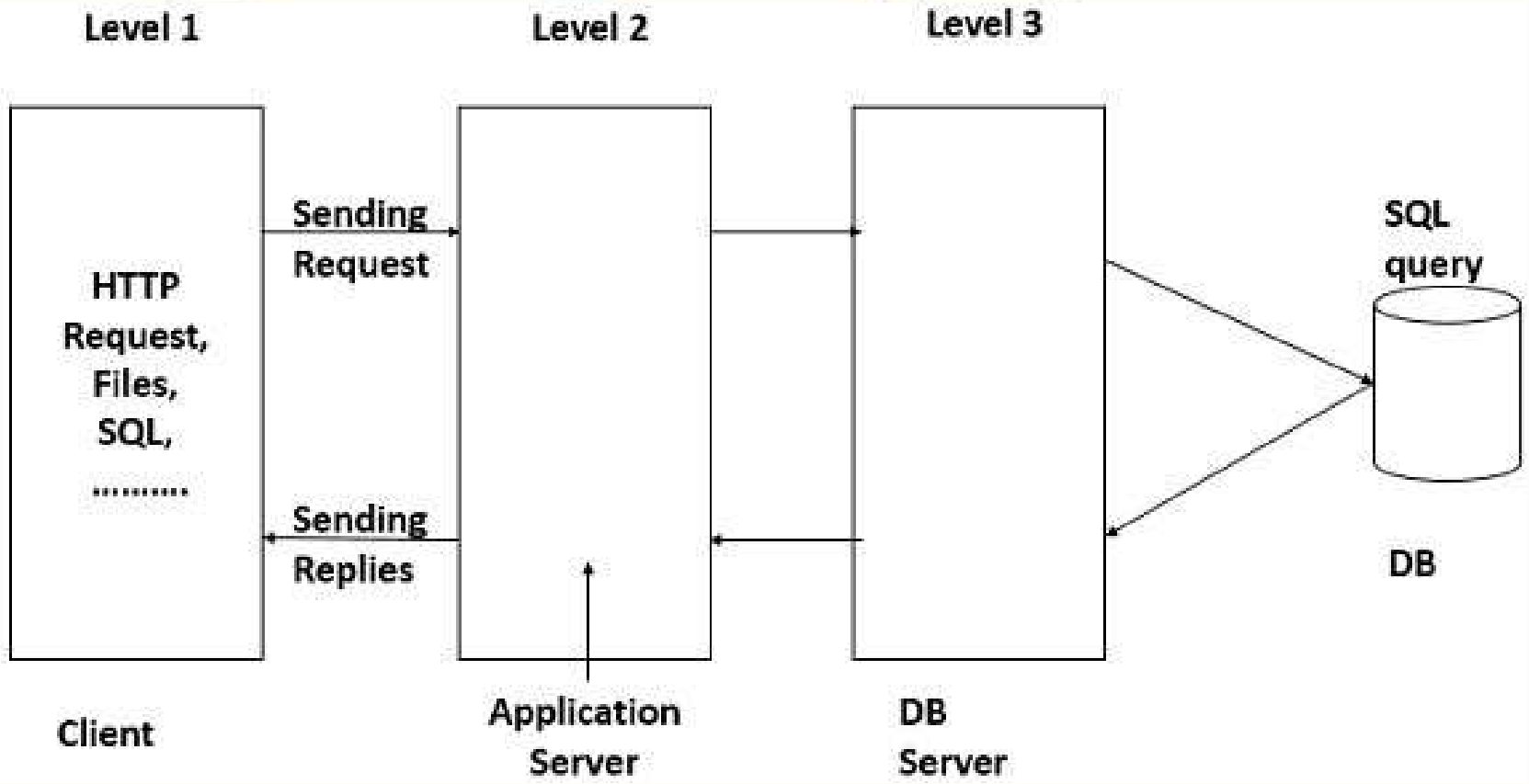


3-Tier Architecture (3TA)

- **M**ost general use of multi-tier architecture
- Typically composed of a ***presentation tier***, an ***application tier***, and a ***data storage*** tier and may execute on a separate processor



3-Tier Architecture (3TA)



Presentation Tier

- The topmost level of the application by which users can access directly such as webpage or Operating System GUI (Graphical User interface)
- The primary function is to translate the tasks and results to something that user can understand
- Communicates with other tiers so that it places the results to the browser/client tier and all other tiers in the network

Application Tier

- **Business Logic, Logic Tier**, or **Middle Tier**
- Coordinates the application, processes the commands, makes logical decisions, evaluation, and performs calculations
- Controls an application's functionality by performing detailed processing
- Moves and processes data between the two surrounding layers

Data Tier

In this layer:

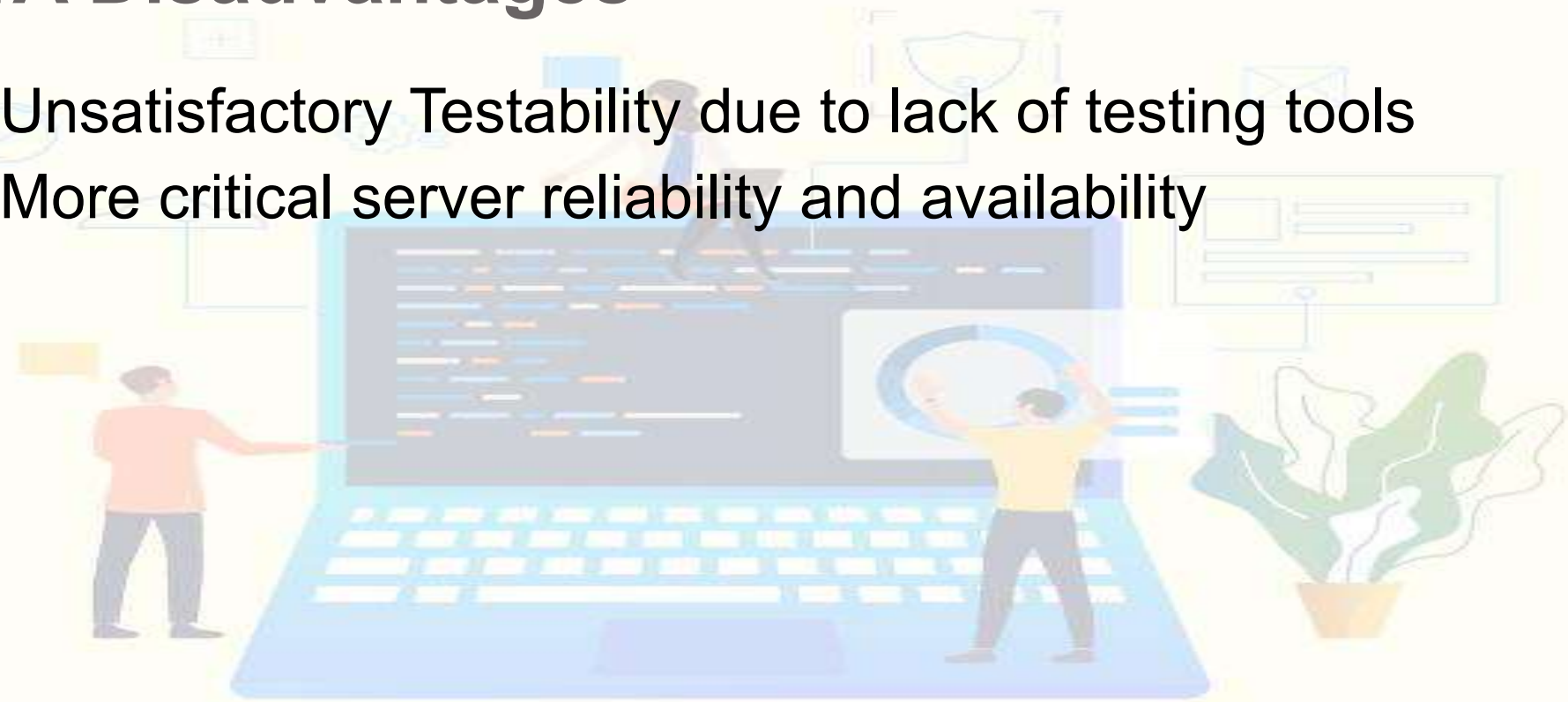
- Information is stored and retrieved from the database or file system
- The information is then passed back for processing and then back to the user.
- It includes the data persistence mechanisms (database servers, file shares, etc.) and provides API (Application Programming Interface) to the application tier which provides methods of managing the stored data

MTA Advantages

- Better performance than a thin-client approach and is simpler to manage than a thick-client approach
- Enhances the reusability and scalability − as demands increase, extra servers can be added
- Provides multi-threading support and also reduces network traffic
- Provides maintainability and flexibility

MTA Disadvantages

- Unsatisfactory Testability due to lack of testing tools
- More critical server reliability and availability



Broker Architectural Style (BAS)

- A middleware architecture used in distributed computing to coordinate and enable the communication between registered servers and clients
  - Here, object communication takes place through a middleware system called an ***object request broker*** (***software bus***)

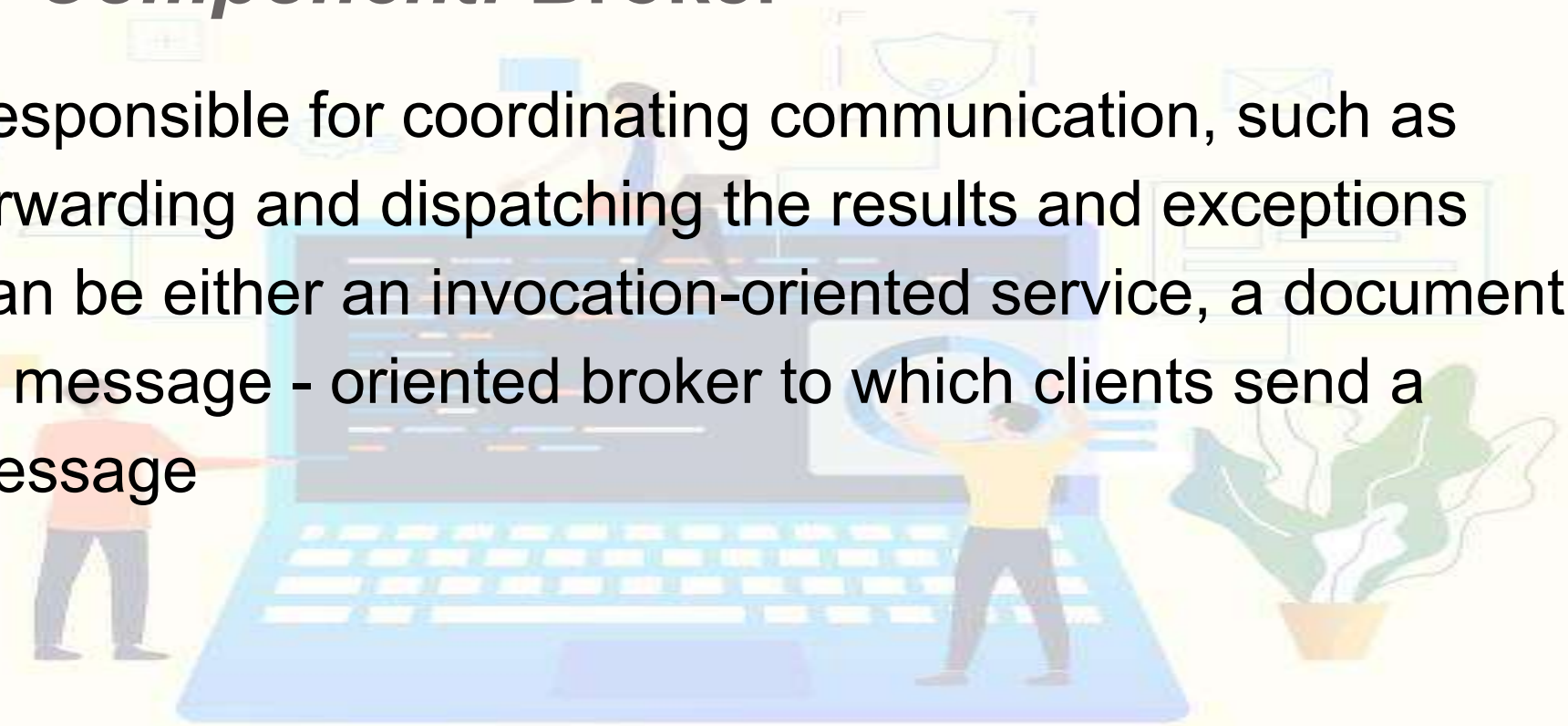
Broker Architectural Style (BAS)

In this style:

- Client and the server do not interact with each other directly
- Client and server have a direct connection to its proxy which communicates with the mediator-broker
- A server provides services by registering and publishing their interfaces with the broker and clients can request the services from the broker statically or dynamically by look-up
- CORBA (Common Object Request Broker Architecture) is a good implementation example of the broker architecture

BAS Component: Broker

- Responsible for coordinating communication, such as forwarding and dispatching the results and exceptions
- Can be either an invocation-oriented service, a document or message - oriented broker to which clients send a message



BAS Component: Broker

- Responsible for brokering the service requests, locating a proper server, transmitting requests, and sending responses back to clients
- Retains the servers' registration information including their functionality and services as well as location information
- Provides APIs for clients to request, servers to respond, registering or unregistering server components, transferring messages, and locating servers

BAS Component: Stub

- Generated at the static compilation time and then deployed to the client side which is used as a proxy for the client
  - Client-side proxy acts as a mediator between the client and the broker and provides additional transparency between them and the client; a remote object appears like a local one
  - The proxy hides the IPC (inter-process communication) at protocol level and performs marshaling of parameter values and unmarshaling of results from the server

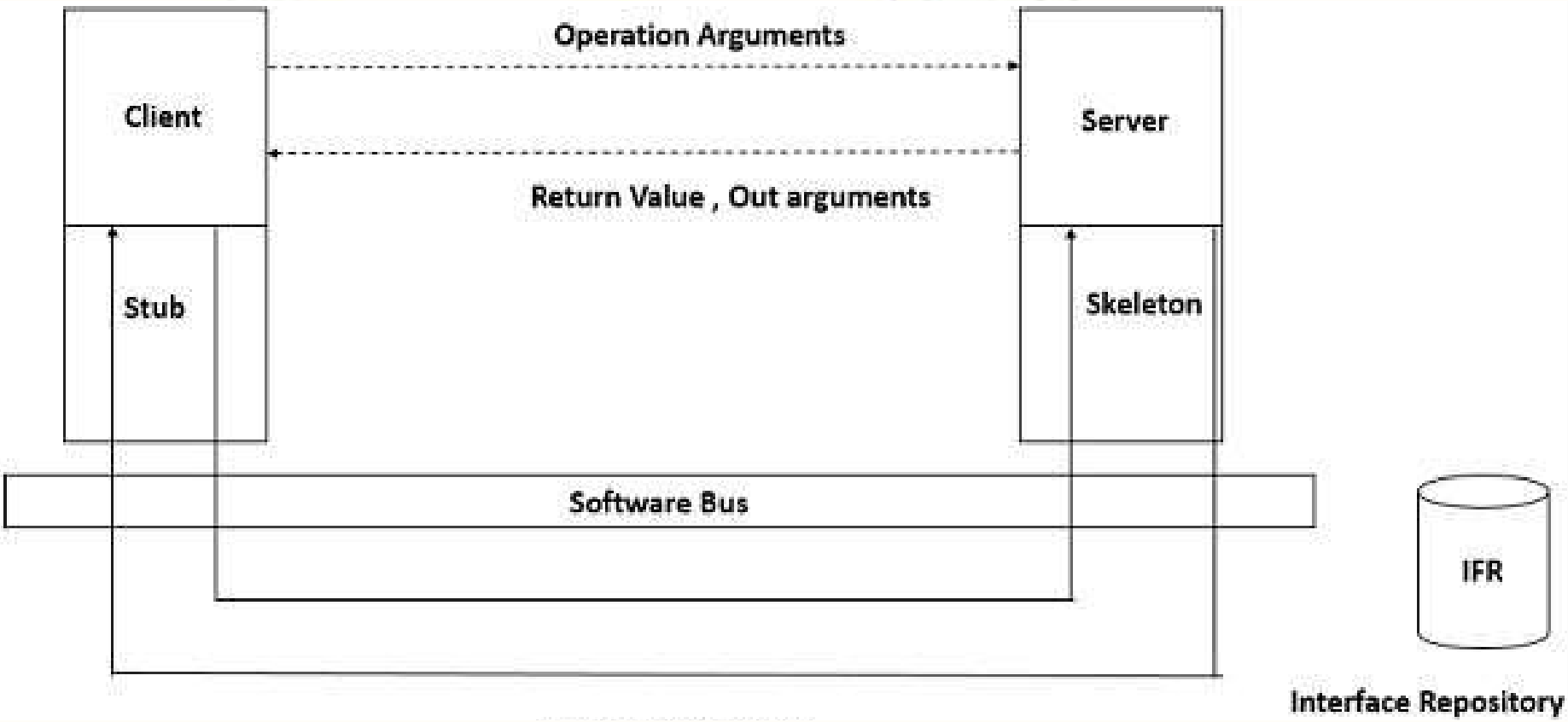
BAS Component: Skeleton

- Generated by the service interface compilation and then deployed to the server side, which is used as a proxy for the server
  - Server-side proxy encapsulates low-level system-specific networking functions and provides high-level APIs to mediate between the server and the broker
- Receives the requests, unpacks the requests, unmarshals the method arguments, calls the suitable service, and also marshals the result before sending it back to the client

BAS Component: Bridge

- Can connect two different networks based on different communication protocols
  - It mediates different brokers including DCOM, .NET remote, and Java CORBA brokers.
- Optional component, which hides the implementation details when two brokers interoperate and take requests and parameters in one format and translate them to another format

BAS Implementation in CORBA



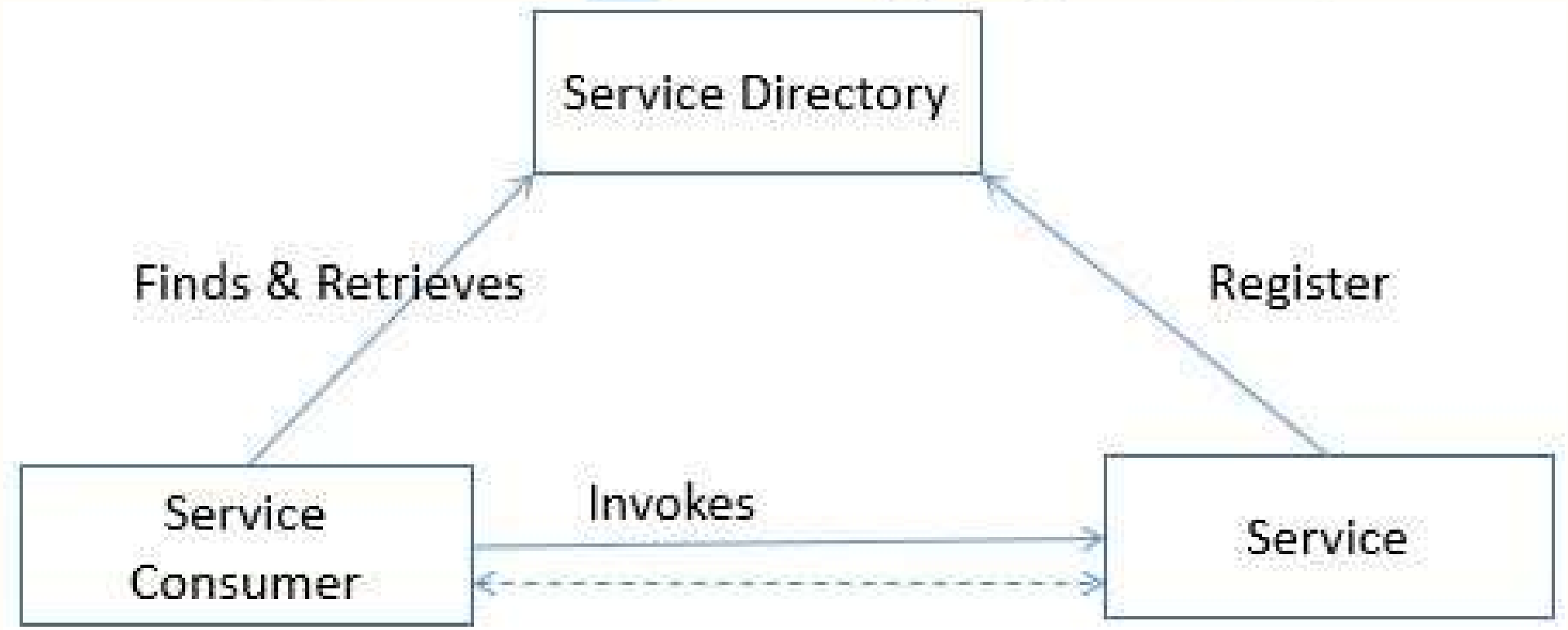
About CORBA

- ***CORBA** is an international standard for an Object Request Broker – a middleware to manage communications among distributed objects defined by Object Management Group (OMG)*

Service-Oriented Architecture (SOA)

- A client/server design which support business-driven IT approach in which an application consists of software ***services*** and ***software service consumers*** (also known as clients or service requesters)

Service-Oriented Architecture (SOA)



Service

- A component of business functionality that is well-defined, self-contained, independent, published, and available to be used via a standard programming interface
  - The connections between services are conducted by common and ***universal message-oriented protocols*** such as the SOAP Web service protocol, which can deliver requests and responses between services loosely

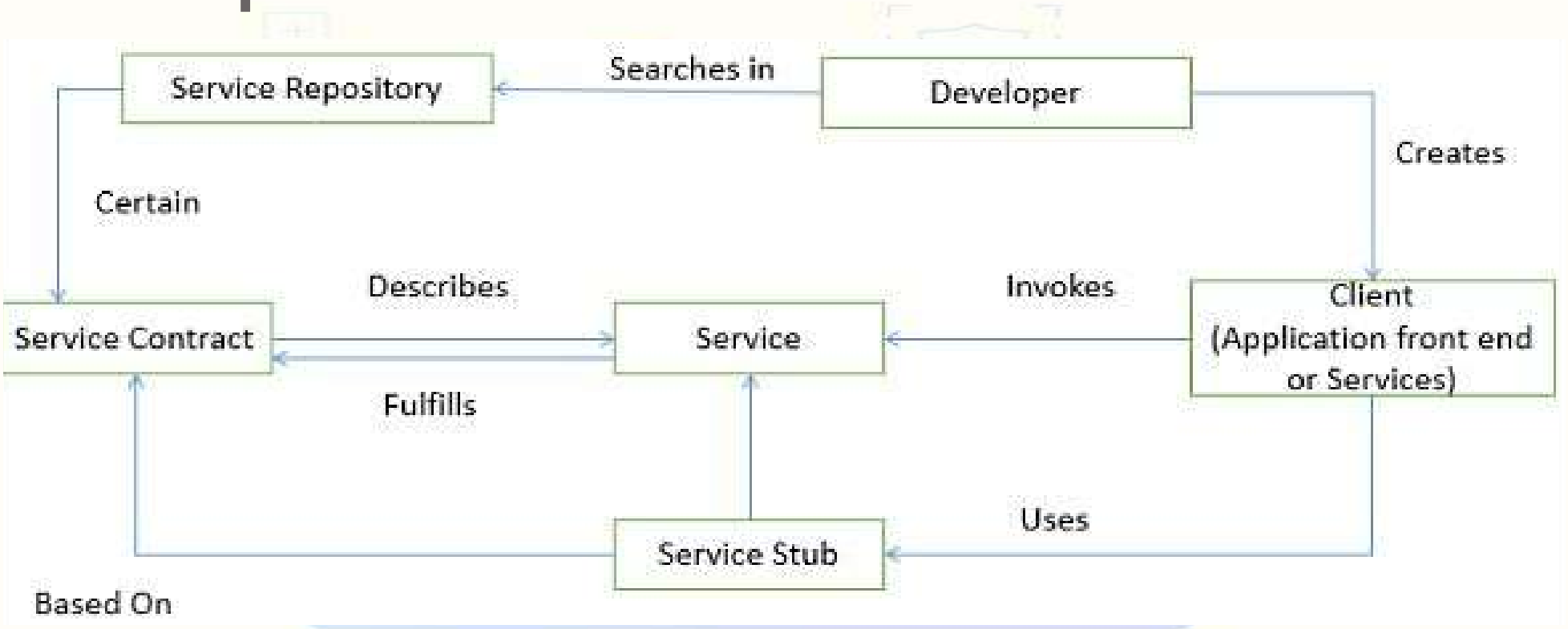
SOA Features

- **Distributed Deployment**: expose enterprise data and business logic as loosely, coupled, discoverable, structured, standard-based, coarse-grained, stateless units of functionality called services
- **Composability**: assemble new processes from existing services that are exposed at a desired granularity through well defined, published, and standard complaint interfaces

SOA Features

- **Interoperability**: share capabilities and reuse shared services across a network irrespective of underlying protocols or implementation technology
- **Reusability**: choose a service provider and access to existing resources exposed as services

SOA Operations



SOA Advantages

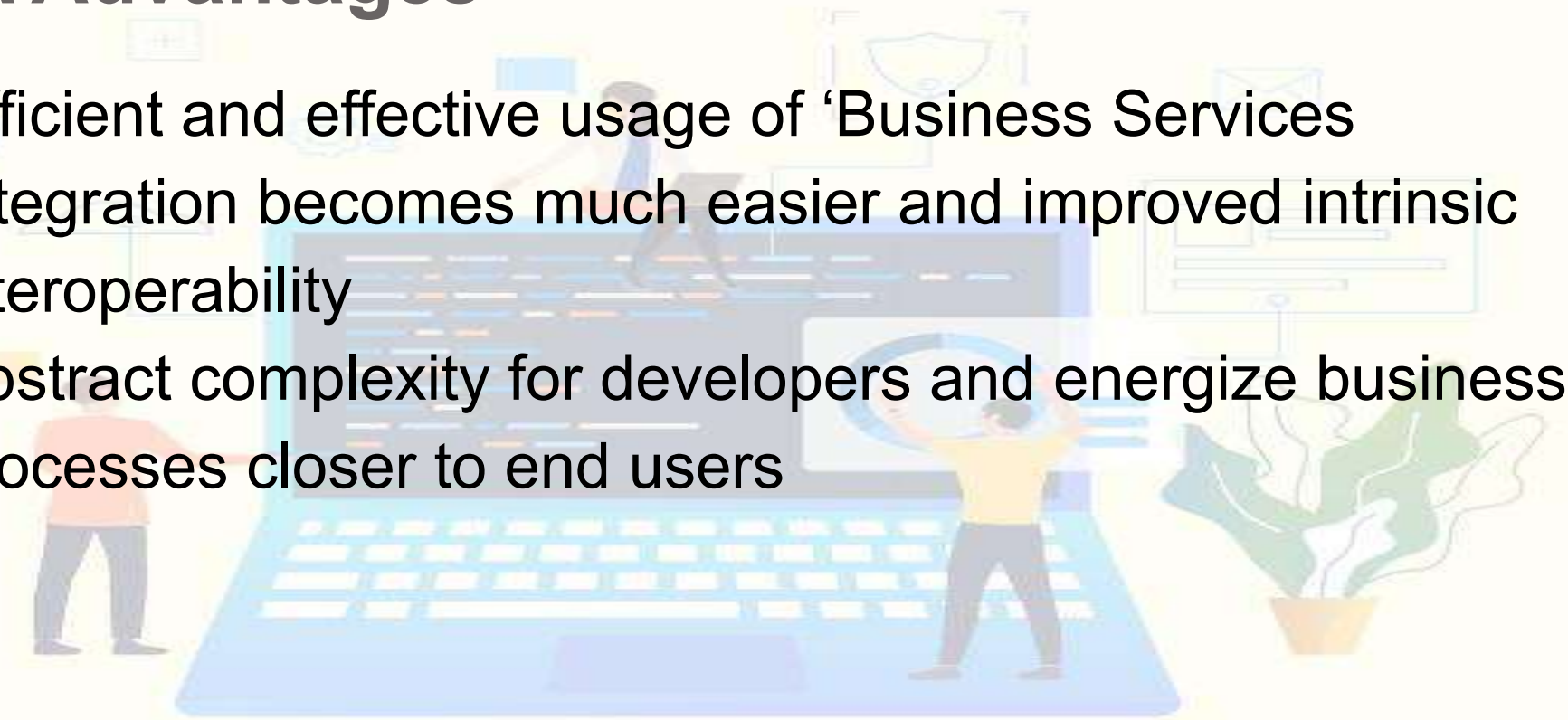
- Loose coupling of service–orientation provides great flexibility for enterprises to make use of all available service recourses irrespective of platform and technology restrictions
- Each service component is independent from other services due to the stateless service feature
- The implementation of a service will not affect the application of the service as long as the exposed interface is not changed

SOA Advantages

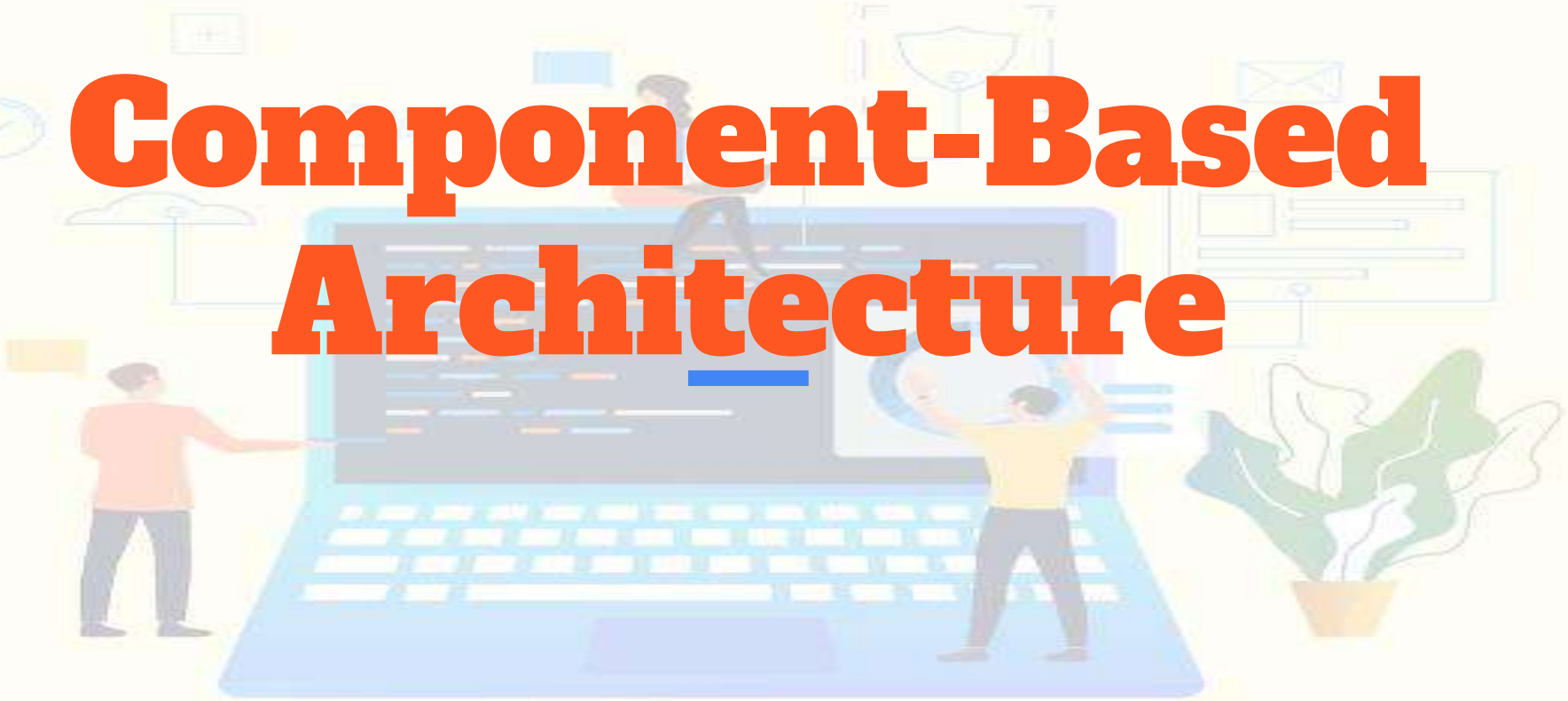
- A client or any service can access other services regardless of their platform, technology, vendors, or language implementations
- Reusability of assets and services since clients of a service only need to know its public interfaces, service composition
- SOA based business application development are much more efficient in terms of time and cost
- Enhances the scalability and provide standard connection

SOA Advantages

- Efficient and effective usage of ‘Business Services
- Integration becomes much easier and improved intrinsic interoperability
- Abstract complexity for developers and energize business processes closer to end users



Component-Based Architecture



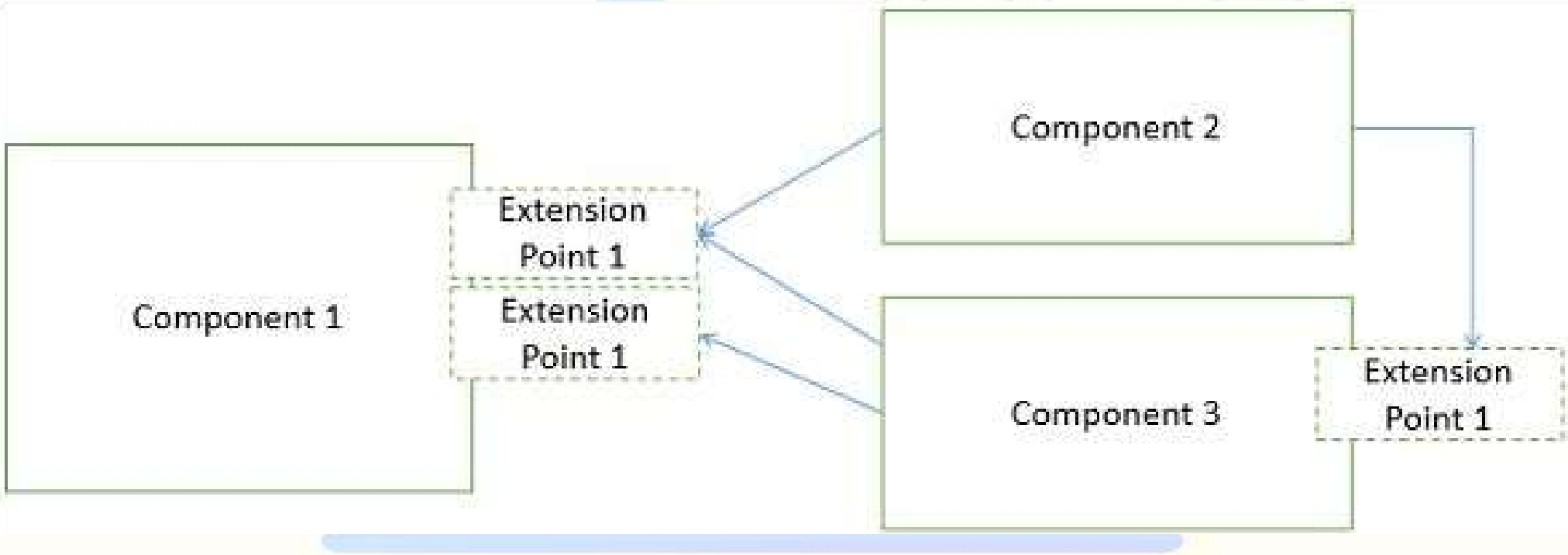
Component-Based Architecture (CBA)

- Focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions

Component-Based Architecture (CBA)

- ***Primary objective***: to ensure ***component reusability***
  - *A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit*
- Standard component frameworks: ***COM/DCOM***, ***JavaBean***, ***EJB***, ***CORBA***, ***.NET***, ***web services***, & ***grid services***
  - *These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation*

Component-Based Architecture (CBA)

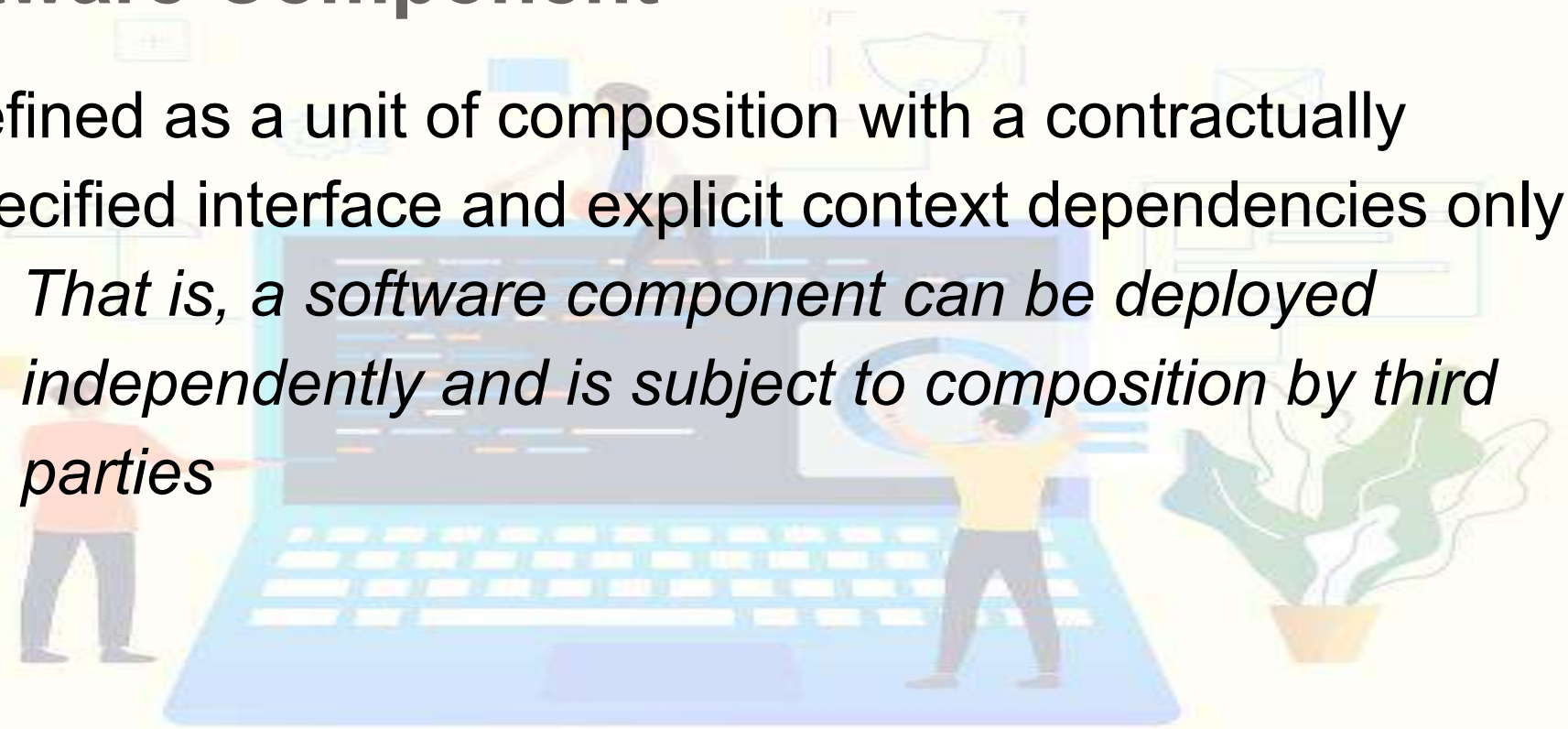


Component

- A modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface
- A software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities
  - It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture

Software Component

- Defined as a unit of composition with a contractually specified interface and explicit context dependencies only
  - *That is, a software component can be deployed independently and is subject to composition by third parties*

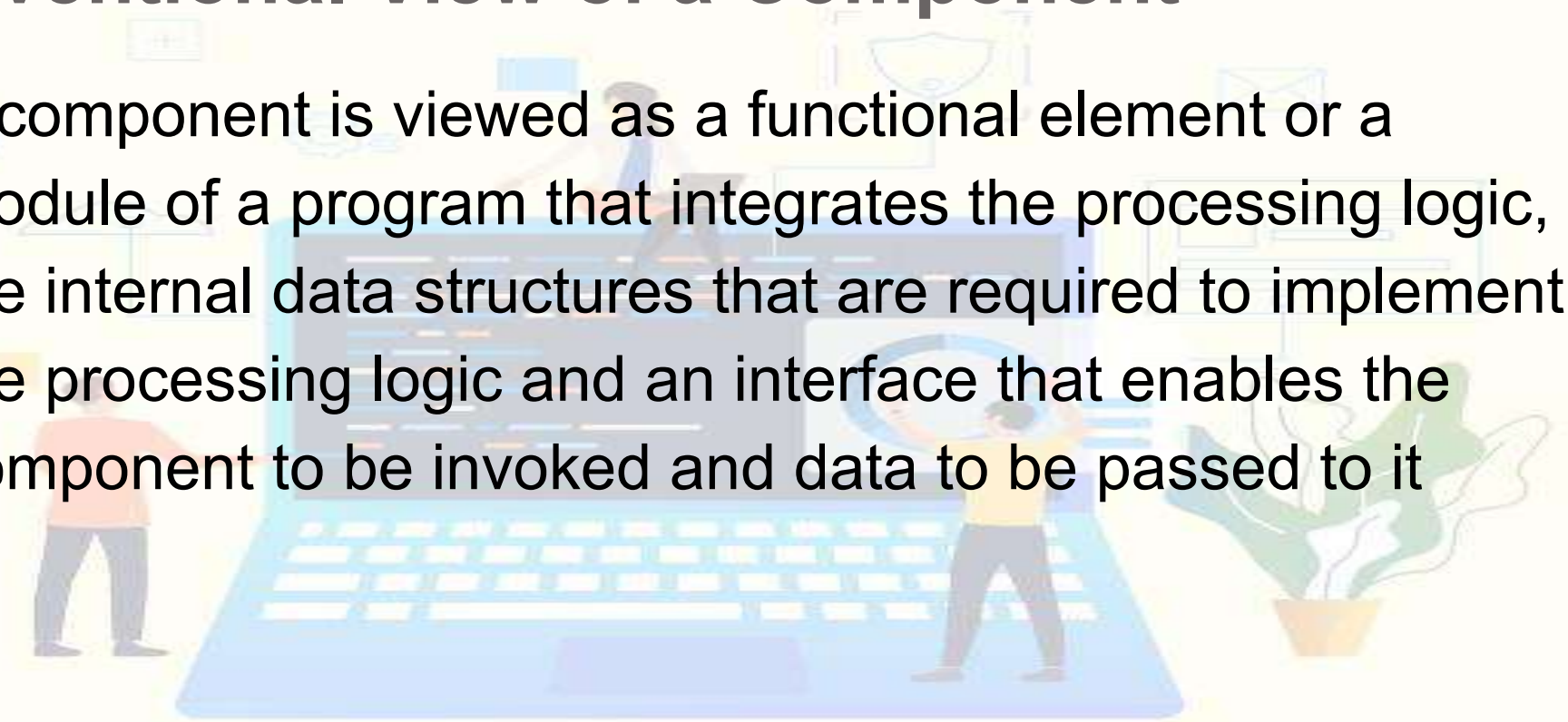


Object-Oriented View of a Component

- A component is viewed as a set of one or more cooperating classes
- Each problem domain class (analysis) and infrastructure class (design) are explained to identify all attributes and operations that apply to its implementation
- It also involves defining the interfaces that enable classes to communicate and cooperate

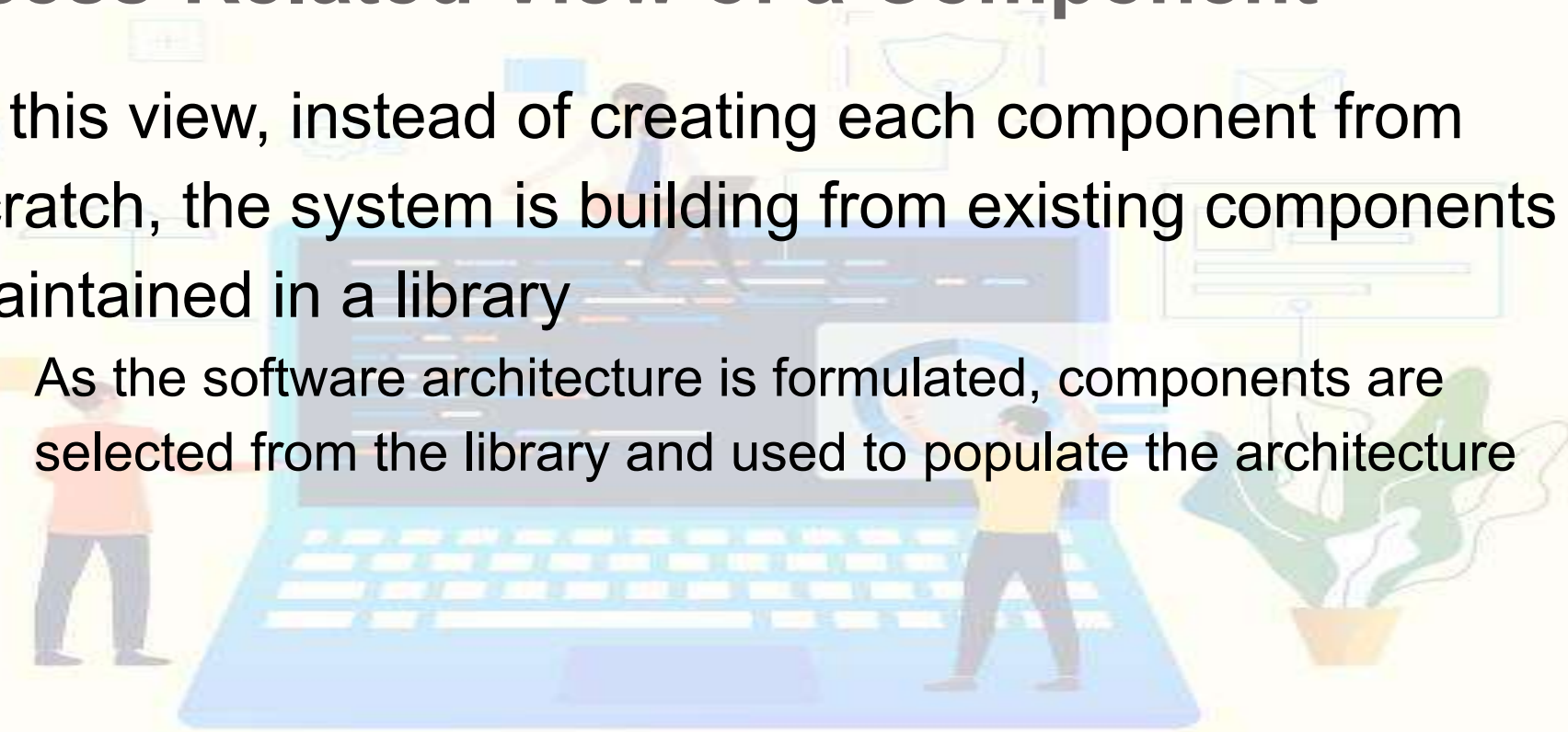
Conventional View of a Component

- A component is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it



Process-Related View of a Component

- In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library
  - As the software architecture is formulated, components are selected from the library and used to populate the architecture



Process-Related View of a Component

In this view:

- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach
- Many components are invisible which are distributed in enterprise business applications and internet web applications (e.g. Enterprise JavaBean (EJB), .NET components, & CORBA components)

Characteristics of Components

- **Reusability**. *Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task*
- **Replaceable**. *Components may be freely substituted with other similar components*
- **Not context specific**. *Components are designed to operate in different environments and contexts*

Characteristics of Components

- **Extensible**. *A component can be extended from existing components to provide new behavior*
- **Encapsulated**. *A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state*
- **Independent**. *Components are designed to have minimal dependencies on other components*

Principles of Component−Based Design

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.

Principles of Component−Based Design

- Depend on abstractions component do not depend on other concrete components, which increase difficulty in expendability
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components.
- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.

Principles of Component−Based Design

- For a server class, specialized interfaces should be created to serve major categories of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.
- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plugin to offer another plugin API.

Component-Level Design Guidelines

- Attains architectural component names from the problem domain and ensures that they have meaning to all stakeholders who view the architectural model.
- Extracts the business process entities that can exist independently without any associated dependency on other entities.
- Recognizes and discover these independent entities as new components.

Component-Level Design Guidelines

- Uses infrastructure component names that reflect their implementation-specific meaning.
- Models any dependencies from left to right and inheritance from top (base class) to bottom (derived classes).
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

Conducting Component-Level Design

- Recognizes all design classes that correspond to the infrastructure domain.
- Describes all design classes that are not acquired as reusable components, and specifies message details.
- Identifies appropriate interfaces for each component and elaborates attributes and defines data types and data structures required to implement them.
- Describes processing flow within each operation in detail by means of pseudo code or UML activity diagrams.

Conducting Component-Level Design

- Describes persistent data sources (databases and files) and identifies the classes required to manage them.
- Develop and elaborates behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class.
- Elaborates deployment diagrams to provide additional implementation detail.

Conducting Component-Level Design

- Demonstrates the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environment.
- The final decision can be made by using established design principles and guidelines. Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model.

CBA Advantages

- **Ease of deployment**. *As new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole*
- **Reduced cost**. *The use of third-party components allows you to spread the cost of development and maintenance*
- **Ease of development**. *Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system*

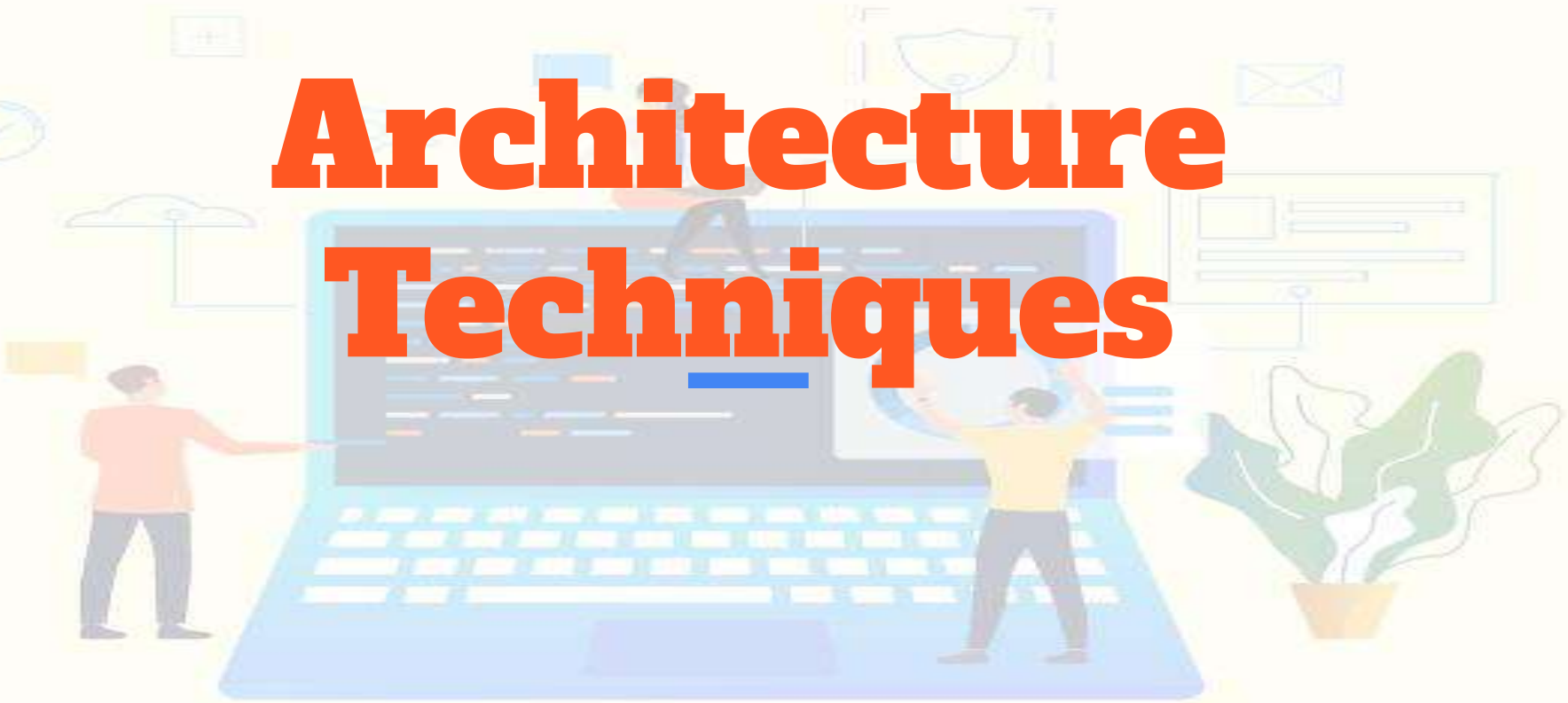
CBA Advantages

- **Reusable**. *The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems*
- **Modification of technical complexity**. *A component modifies the complexity through the use of a component container and its services*
- **Reliability**. *The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse*

CBA Advantages

- **System maintenance and evolution**. *Easy to change and update the implementation without affecting the rest of the system*
- **Independent**. *Independency and flexible connectivity of components*
  - *Independent development of components by different group in parallel*
  - *Productivity for the software development and future software development*

Architecture Techniques



Iterative and Incremental Approach

- It is an iterative and incremental approach consisting of five main steps that helps to generate candidate solutions
- This candidate solution can further be refined by repeating these steps and finally create an architecture design that best fits our application
- At the end of the process, we can review and communicate our architecture to all interested parties
- There are many other more formal approaches that defining, reviewing, and communicating your architecture

Identify Architecture Goal

- Identify the architecture goal that forms the architecture and design process
- Flawless and defined objectives emphasize on the architecture, solve the right problems in the design and helps to determine when the current phase has completed, and ready to move to the next phase
- ***Activities:***
  - Identify your architecture goals at the start.
  - Identify the consumer of our architecture.
  - Identify the constraints.

Key Scenarios

- This step puts emphasis on the design that matters the most
- A ***scenario*** is an extensive and covering description of a user's interaction with the system
- ***Key scenarios*** are those that are considered the most important scenarios for the success of your application
  - It helps to make decisions about the architecture
  - The goal is to achieve a balance among the user, business, and system objectives.*(e.g., user authentication is a key scenario because they are an intersection of a quality attribute (security) with important functionality (how a user logs into your system))*

Application Overview

Build an overview of application, which makes the architecture more touchable, connecting it to real-world constraints and decisions

Identify Application Type

- Identify application type whether it is a mobile application, a rich client, a rich internet application, a service, a web application, or some combination of these type

Application Overview

Identify Deployment Constraints

- Choose an appropriate deployment topology and resolve conflicts between the application and the target infrastructure

Identify Important Architecture Design Styles

- Identify important architecture design styles such as client/server, layered, message-bus, domain-driven design, etc. to improve partitioning and promotes design reuse by providing solutions to frequently recurring problems.
- Applications will often use a combination of styles

Application Overview

Identify the Relevant Technologies

- Identify the relevant technologies by considering the type of application we are developing, our preferred options for application deployment topology and architectural styles
- The choice of technologies will also be directed by organization policies, infrastructure limitations, resource skills, and so on

Application Overview

Key Issues or Key Hotspots

- While designing an application, hot spots are the zones where mistakes are most often made. Identify key issues based on quality attributes and crosscutting concerns
- Potential issues include the appearance of new technologies and critical business requirements

Application Overview

Key Issues or Key Hotspots

- Quality attributes are the overall features of your architecture that affect run-time behavior, system design, and user experience. Crosscutting concerns are the features of our design that may apply across all layers, components, and tiers
- These are also the areas in which high-impact design mistakes are most often made. Examples of crosscutting concerns are authentication and authorization, communication, configuration management, exception management and validation, et

Application Overview

Candidate Solutions

- After defining the key hotspots, build the initial baseline architecture or first high level design and then start to fill in the details to generate candidate architecture
- Candidate architecture includes the application type, the deployment architecture, the architectural style, technology choices, quality attributes, and crosscutting concerns. If the candidate architecture is an improvement, it can become the baseline from which new candidate architectures can be created and tested

Application Overview

Candidate Solutions

- Validate the candidate solution design against the key scenarios and requirements that have already defined, before iteratively following the cycle and improving the design
- May use architectural spikes to discover the specific areas of the design or to validate new concepts
  - Architectural spikes are a design prototype, which determine the feasibility of a specific design path, reduce the risk, and quickly determine the viability of different approaches
  - Test architectural spikes against key scenarios and hotspots

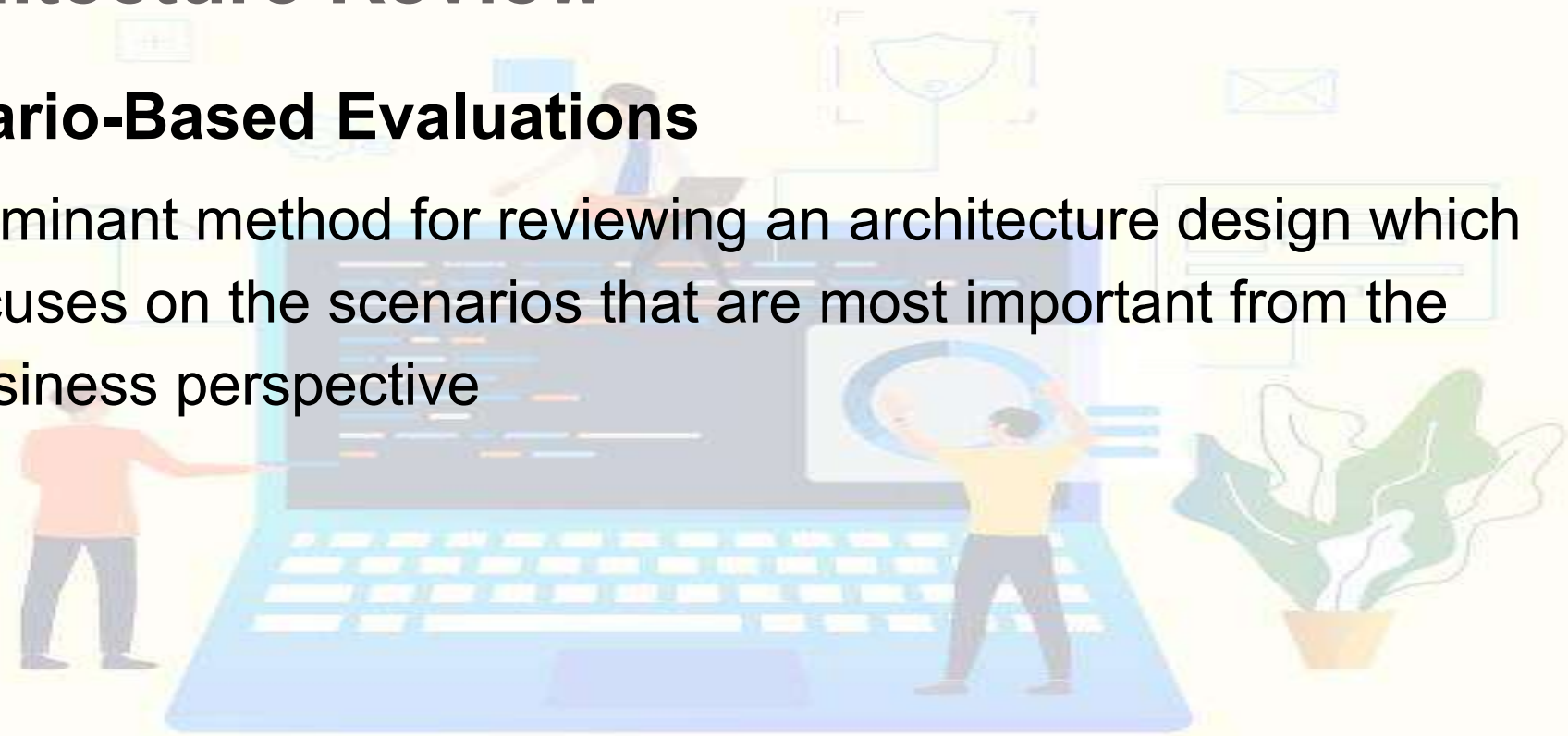
Architecture Review

- Most important task in order to reduce the cost of mistakes and to find and fix architectural problems as early as possible.
- Well-established, cost-effective way of reducing project costs and the chances of project failure
  - *Review the architecture frequently at major project milestones, and in response to other significant architectural changes*
  - *The main objective of an architecture review is to determine the feasibility of baseline and candidate architectures, which verify the architecture correctly*
  - *Links the functional requirements and the quality attributes with the proposed technical solution*
  - *It also helps to identify issues and recognize areas for improvement*

Architecture Review

Scenario-Based Evaluations

- Dominant method for reviewing an architecture design which focuses on the scenarios that are most important from the business perspective



Architecture Review Methodologies

- **Software Architecture Analysis Method (SAAM)**
  - It is originally designed for assessing modifiability, but later was extended for reviewing architecture with respect to quality attributes.
- **Architecture Tradeoff Analysis Method (ATAM)**
  - It is a polished and improved version of SAAM, which reviews architectural decisions with respect to the quality attributes requirements, and how well they satisfy particular quality goals

Architecture Review Methodologies

- **Active Design Review (ADR)**
  - It is best suited for incomplete or in-progress architectures, which more focus on a set of issues or individual sections of the architecture at a time, rather than performing a general review
- **Active Reviews of Intermediate Designs (ARID)**
  - It combines the ADR aspect of reviewing in-progress architecture with a focus on a set of issues, and the ATAM and SAAM approach of scenario-based review focused on quality attributes

Architecture Review Methodologies

- **Cost Benefit Analysis Method (CBAM)**
  - It focuses on analyzing the costs, benefits, and schedule implications of architectural decisions
- **Architecture Level Modifiability Analysis (ALMA)**
  - It estimates the modifiability of architecture for business information systems (BIS)
- **Family Architecture Assessment Method (FAAM)**
  - It estimates information system family architectures for interoperability and extensibility

Communicating the Architecture Design

- **4 + 1 Model**
  - This approach uses five views of the complete architecture. Among them, four views (the logical view, the process view, the physical view, and the development view) describe the architecture from different approaches
  - A fifth view shows the scenarios and use cases for the software. It allows stakeholders to see the features of the architecture that specifically interest them

Communicating the Architecture Design

- **Architecture Description Language (ADL)**
  - This approach is used to describe software architecture prior to the system implementation
  - It addresses the following concerns – behavior, protocol, and connector
  - The main advantage of ADL is that we can analyze the architecture for completeness, consistency, ambiguity, and performance before formally beginning use of the design

Communicating the Architecture Design

- **Agile Modeling**
  - This approach follows the concept that “content is more important than representation.” It ensures that the models created are simple and easy to understand, sufficiently accurate, detailed, and consistent
  - Agile model documents target specific customer(s) and fulfill the work efforts of that customer. The simplicity of the document ensures that there is active participation of stakeholders in the modeling of the artifact

Communicating the Architecture Design

- **IEEE 1471**
  - IEEE 1471 is the short name for a standard formally known as ANSI/IEEE 1471-2000, “Recommended Practice for Architecture Description of Software-Intensive Systems.”
  - IEEE 1471 enhances the content of an architectural description, in particular, giving specific meaning to context, views, and viewpoints

Communicating the Architecture Design

- **Unified Modeling Language (UML)**
  - This approach represents three views of a system model
  - The functional requirements view (functional requirements of the system from the point of view of the user, including use cases); the static structural view (objects, attributes, relationships, and operations including class diagrams); and the dynamic behavior view (collaboration among objects and changes to the internal state of objects, including sequence, activity, and state diagrams).

Software Architecture & Design