
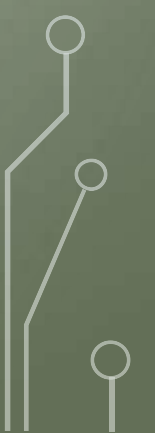
A decorative graphic on the left side of the slide, consisting of a network of thin, light blue lines that resemble a circuit board or a neural network. These lines are interconnected with small, light blue circles, creating a complex, branching pattern that extends from the top left towards the bottom left.

CHAPTER 3 - KEY ISSUES IN SOFTWARE DESIGN



LECTURE OUTLINE

- Overview of key issues in software design
 - Concurrency
 - Control and Handling of Events
 - Data Persistence
 - Distribution of Components
 - Error and Exception Handling and Fault Tolerance
 - Interaction and Presentation
 - Security
- 
- 

OVERVIEW OF KEY ISSUES IN SOFTWARE DESIGN

- A number of key issues must be dealt with when designing software.
- Some are quality concerns that all software must address
 - for example, performance, security, reliability, usability, etc.
- Another important issue is how to decompose, organize, and package software components.
- This is so fundamental that all design approaches address it in one way or another.

OVERVIEW OF KEY ISSUES IN SOFTWARE DESIGN



- In the analysis of the problem domain and structuring a system into subsystems, the emphasis is on functional decomposition:
 - Traditional software development
 - Each subsystem addresses a distinctly separate part of the system. A subsystem can be structured further into smaller subsystems.
 - The design goal – to perform a major function that is relatively independent of the functionality provided by other subsystems.
 - Top-down or bottom-up

OVERVIEW OF KEY ISSUES IN SOFTWARE DESIGN

- In contrast, other issues “deal with some aspect of software’s behavior that is **not** in the application domain, but which addresses some of the supporting domains”.
- Such issues, which often crosscut the system’s functionality, have been referred to as **aspects**, which “tend not to be units of software’s functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways”. (Aspect-oriented software development).



KEY ISSUES

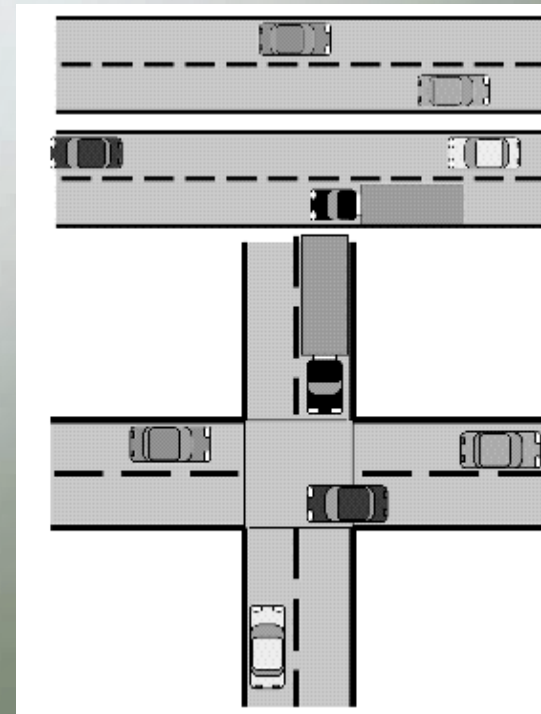
- Concurrency
 - Control and Handling of Events
 - Data Persistence
 - Distribution of Components
 - Error and Exception Handling and Fault Tolerance
 - Interaction and Presentation
 - Security
- 
- 

A decorative graphic on the left side of the slide, consisting of a network of thin, light blue lines that resemble a circuit board or a neural network. These lines are connected to small, empty circles at various points, creating a complex, branching pattern that extends from the top to the bottom of the frame.

CONCURRENCY

CONCURRENCY

- Concurrency is the tendency for things to happen at the same time in a system.
- Concurrency is a natural phenomenon, of course. In the real world, at any given time, many things are happening simultaneously.
- When we design software to monitor and control real-world systems, we must deal with this natural concurrency.



CONCURRENCY


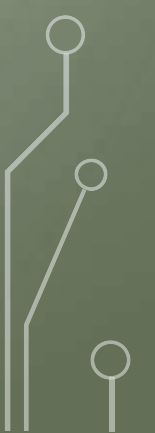
- In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.
- It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.
- Example : The spell check feature in word processor is a module of software, which runs along side the word processor itself.

CONCURRENCY

- When dealing with concurrency issues in software systems, there are generally two aspects that are important:
 - being able to detect and respond to external events occurring in a random order.
 - ensuring that these events are responded to in some minimum required interval.



CONCURRENCY

- If each concurrent activity evolved independently, in a truly parallel fashion, this would be relatively simple: we could simply create separate programs to deal with each activity.
 - The challenges of designing concurrent systems arise mostly because of the interactions which happen between concurrent activities. When concurrent activities interact, some sort of coordination is required.
- 
- 

CONCURRENCY


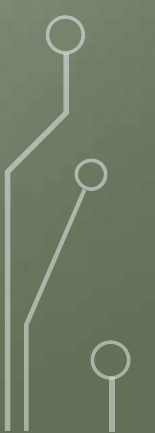
- The success of a concurrent system depends on well designed hardware, flexible software that controls the hardware, and clear marketing vision.
- To adapt the changing marketing requirement, hardware and software need to have flexible architectures.
- Any multi-threaded system can be considered a concurrent system;
 - For example, a lengthy task can be implemented as a background thread so that it will not block the graphical user interface.

CONCURRENCY

- Concurrent systems can be characterized by the following traits:
 - System input and output can be clearly identified.
 - System internal consists of system resources, such as hardware modules, which are used to process system input and generates system output.
 - One or more execution steps are needed for a system input to be processed by system resources and to become a system output.
 - The system resources and their relationship are identified by system analysis. The concurrency properties of system resources determine the constraints between execution steps, which ultimately define the system concurrency behaviour.

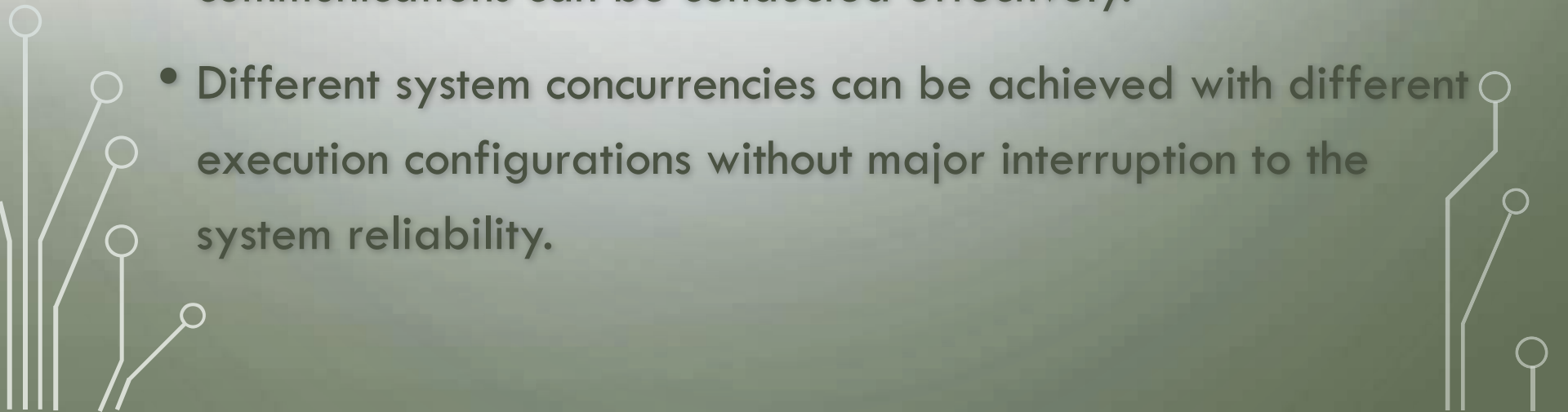


CONCURRENCY DESIGN GOALS

- Have an easy to understand software architecture so that the desired concurrency can be implemented and verified quickly.
 - Have a solid system concurrency kernel to adapt system environmental changes such as inconsistent hardware responses, and still achieve high system reliability.
 - Have a good scalable architecture to adapt new requirement changes.
- 
- 



CONCURRENCY DESIGN GOALS


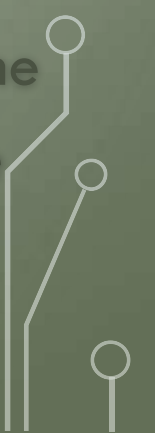
- System concurrency and throughput are well understood by all teams involved in the system specification and design, not just by a few key software engineers. Therefore, the concurrent software should expose how the current system internally works with minimum cost, so that the team communications can be conducted effectively.
 - Different system concurrencies can be achieved with different execution configurations without major interruption to the system reliability.
- 

CONCURRENCY COMMON ISSUES

- The software team member is not very experienced in concurrency design.
 - Most teams have engineers knowing threads, critical sections, semaphores, and events. But, this usually does not guarantee achieving the design goals.
- The understanding of system concurrency is very slow.
 - The software engineer could not present a full picture of how the system concurrency design is going to be working until the alpha or beta stage.



CONCURRENCY COMMON ISSUES

- The marketing group has a wrong system throughput assumption and commitment at the beginning of a project, with false understanding of the system resource constraints, or the complexity for available engineers to achieve the desired high throughput without sacrificing software system reliability.
 - Almost all designs do not have a clear distinction between the code controlling the system resource operation and the code performing the system resource concurrency.
- 
- 

CONCURRENCY COMMON ISSUES

- The fragile concurrency architecture is hard to understand.
 - It is almost impossible for new engineers to take over the design, except to abandon the old one, and then propose a "better architecture" which usually goes through the same design cycle and delays the schedule.
- The manager and software engineer mistakenly think that object oriented analysis of the concurrent hardware modules will guarantee a good concurrent software design which delivers a flexible concurrent software architecture.

A decorative graphic on the left side of the slide consisting of a network of thin, light blue lines. These lines form a complex, branching pattern that resembles a circuit board or a neural network. Some lines end in small circles, while others are open. The pattern is denser in the upper left and tapers off towards the bottom.

CONTROL AND HANDLING OF EVENTS

CONTROL AND HANDLING OF EVENTS

- This design issue is concerned with how to organize data and control flow as well as how to handle reactive and temporal events through various mechanisms such as **implicit invocation** and **callbacks**.

■ A reactive system is a system that responds (reacts) to external events (input events).


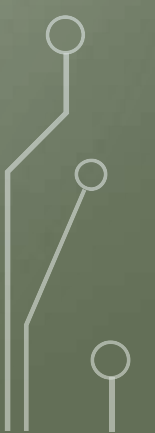
- For example, a light consisting of a bulb and a switch is a reactive system, reacting to the user changing the switch position

■ A temporal events is a time-based system; event handling triggered at a specific time.



CONTROL AND HANDLING OF EVENTS

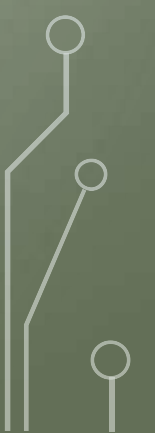

- **Implicit invocation**

- an example of a well-crafted architectural style with high cohesion and loose coupling.
 - Implicit invocation systems are driven by events.
 - Events are triggered whenever the system needs to do something - such as respond to an incoming request.
 - Events can take many forms across different types of implementations.
- 
- 



CONTROL AND HANDLING OF EVENTS

“The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement implicitly causes the invocation of procedures in other modules.” — David Garlan and Mary Shaw



CONTROL AND HANDLING OF EVENTS

- **Callback**

- A callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time.
- The invocation may be immediate as in a synchronous callback or it might happen at later time, as in an asynchronous callback.
- In all cases, the intention is to specify a function or subroutine as an entity that is, depending on the language, more or less similar to a variable.

CONTROL AND HANDLING OF EVENTS

- Callbacks are used to program applications in windowing systems.
- In this case, the application supplies (a reference to) a specific custom callback function for the operating system to call.
- The OS then calls this application-specific function in response to events like mouse clicks or key presses.




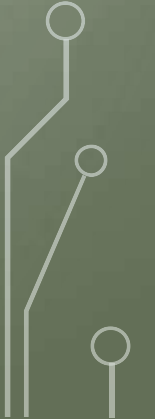
A callback is often back on the level of the original caller.

A decorative graphic on the left side of the slide, consisting of a network of thin, light blue lines that branch out and connect to small circles, resembling a circuit board or a data network. The lines are more dense on the left and become sparser as they move towards the center.

DATA PERSISTENCE


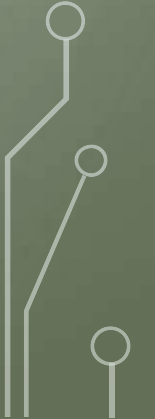


DATA PERSISTENCE

- This design issue is concerned with how to handle long-lived data (database design).
 - Data:
 - Data are the most stable part of an organization's information system.
 - Permanent data are stored in tables within a database
 - Permanent storage of data is also referred to as persistent data.
- 
- 



DATA PERSISTENCE


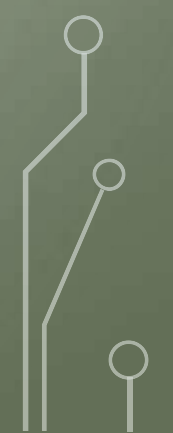
- Why do we need database design?
 - A quality information system demands a quality database design.
 - Avoid redundancy (duplication) of data.
 - Insures simple database structures which allow for maximum effective utilization of the data.
- 
- 

A decorative graphic on the left side of the slide, consisting of a network of thin, light blue lines that branch out and connect to small circles, resembling a circuit board or a neural network. The lines are more dense on the left and become sparser as they move towards the right.

DISTRIBUTION OF COMPONENTS



DISTRIBUTION OF COMPONENTS

- This design issue is concerned with how to distribute the software across the hardware (including computer hardware and network hardware), how the components communicate, and how middleware can be used to deal with heterogeneous software.
- 
- 

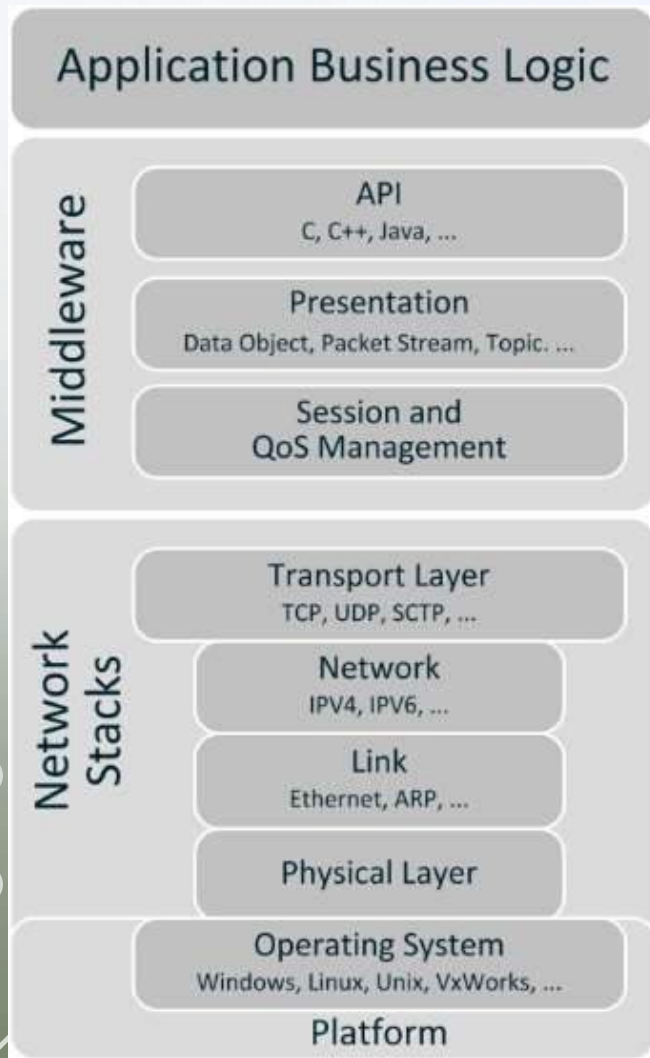
MIDDLEWARE

- Middleware is a software layer situated between applications and operating systems and enables the various components of a system to communicate and manage data.
- Middleware is typically used in distributed systems where it simplifies software development by doing the following:
 - Hides the intricacies (complexities) of distributed applications
 - Hides the heterogeneity of hardware, operating systems and protocols
 - Provides uniform and high-level interfaces used to make interoperable, reusable and portable applications
 - Provides a set of common services that minimizes duplication of efforts and enhances collaboration between applications

MIDDLEWARE

- The purpose of middleware is to simplify the development of complex distributed systems by making it easier for applications to communicate with each other in a networked environment while enabling software developers to focus on the specific purpose of the application (the business logic).
- Middleware is often referred to as "plumbing" as it is used to connect applications and enable them to exchange data. It includes web servers, application servers, object request brokers, messaging systems, data distribution services and similar tools that can support distributed application development and deployment.

MIDDLEWARE



- Middleware can perform a critical role in enabling interoperability between applications that may be written in different programming languages, running on different operating systems and underlying hardware by providing the ability to exchange data in a standards-based way.

TYPES OF MIDDLEWARE

- Message Oriented Middleware (MOM)
 - Supports message exchange
- Remote Procedure Call (RPC)
 - Procedural Middleware
- Object Request Brokers (ORBs)
 - OO version of procedural middleware
- SQL-oriented Middleware
 - Database Middleware
- Transaction Middleware
 - Supports transactions

MESSAGE ORIENTED MIDDLEWARE (MOM)



- A category of communication software that generally relies on asynchronous message-passing in contrast to a request-response style architecture.
- Typically in a message oriented middleware system an extra component called a message broker is used as the transfer agent between sender and receiver.
- The message broker provides important functions such as routing and message queuing (both transient and persistent).

MESSAGE ORIENTED MIDDLEWARE (MOM)

- Message oriented middleware enables users to build "loosely coupled" systems that can evolve easily as sender and receiver applications do not necessarily have knowledge about each other at "compile time" or even be connected to the network at the same time when in operation.
- Examples of MOM technologies and standards include Object Management Group Data Distribution Service for Real-Time Systems (OMG DDS), Java Message Service (JMS), Advanced Message Queuing Protocol (AMQP), and Message Queueing Telemetry Transport (MQTT).



REMOTE PROCEDURE CALL (RPC)

- Is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on network) without the programmer explicitly coding the details for this remote interaction.
 - RPCs are used in systems that require a request-response style architecture.
 - It calls procedures on remote systems and is used to perform synchronous or asynchronous interactions between applications or systems. It is usually utilized within a software application.
- 
- 

REMOTE PROCEDURE CALL (RPC)


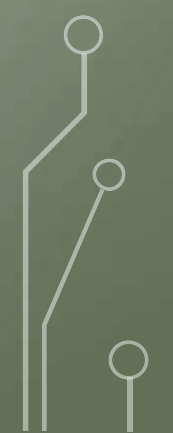
- An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process.
- Examples of RPC technologies include ONC RPC formerly SUN's RPC and XML RPC.

OBJECT REQUEST BROKERS (ORBS)

- In object oriented systems an ORB is a middleware which allows remote procedure calls to be made between objects running in different address spaces over a computer network.
- The whole process is transparent, resulting in remote objects appearing as if they were local. Again ORB middleware is used in systems that require request-response style interactions.
- Request-response based systems introduce a tighter coupling between client and server applications in comparison to Message Oriented Middleware and usually require "compile-time" knowledge of each other in order to communicate.



OBJECT REQUEST BROKERS (ORBS)

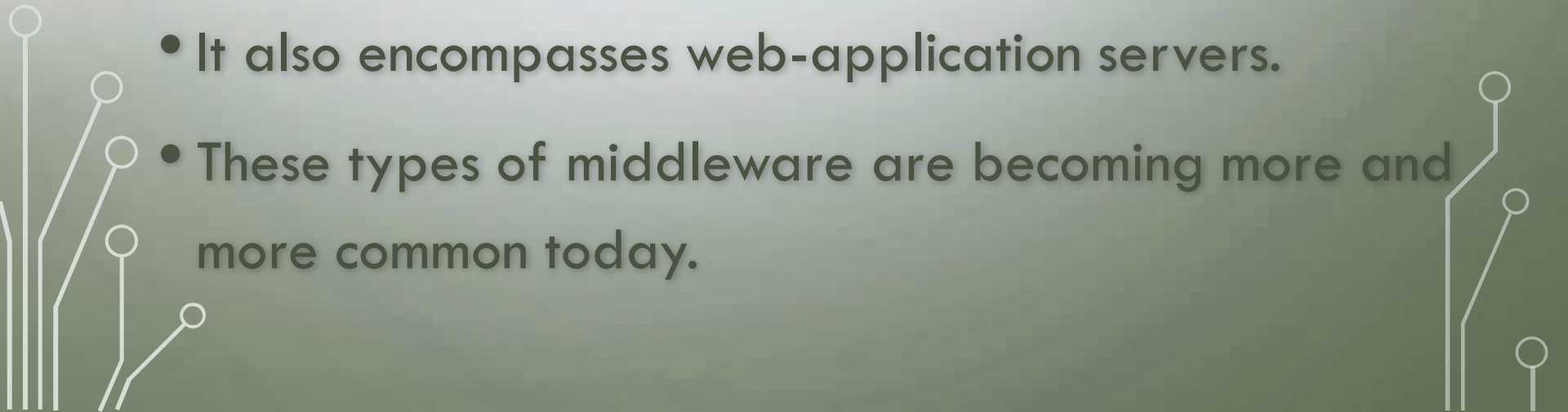
- Examples of ORB technologies include CORBA, .Net Remoting and Java RMI, although the term ORB has become synonymous with the Object Management Group's CORBA (Common Object Request Broker) standard.
- 
- 

SQL-ORIENTED MIDDLEWARE

- This type of middleware allows for direct access to databases, providing direct interaction with them. There are many database gateways and connectivity options.
- Provides transparent access to database objects (Tables, Indexes, Sequences, Views, Synonyms) regardless of their location via an SQL API.
- The database server can reside on the same node as the SQL client or on a different node on the network.



TRANSACTION MIDDLEWARE

- Supports transactions involving components on distributed relational databases.
 - Includes applications like transaction processing monitors.
 - It also encompasses web-application servers.
 - These types of middleware are becoming more and more common today.
- 

A decorative graphic on the left side of the slide, consisting of a network of thin, light blue lines that resemble a circuit board or a neural network. These lines are connected to small, light blue circles, creating a complex, branching pattern that extends from the top left towards the bottom left.

ERROR AND EXCEPTION HANDLING AND FAULT TOLERANCE

ERROR AND EXCEPTION HANDLING AND FAULT TOLERANCE


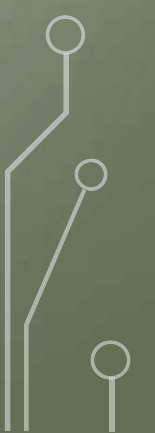
- This design issue is concerned with how to prevent, tolerate, and process errors and deal with exceptional conditions.
- Exception handling differs from fault tolerance, but they are related.
- **Exception handling** can be seen as fault avoidance or fault containment. Unexpected conditions must be masked before they can cause a fault in the system.
 - Deals with the handling of well-defined error conditions within a well-defined systems or framework.

ERROR AND EXCEPTION HANDLING AND FAULT TOLERANCE

- It is not possible to cover every exception within a closed system. There are unanticipated situations that the system cannot compensate for.
 - It is unrealistic to cover all exceptional conditions because they are not predictable.
 - For example, if you only look at the software, environmental exceptional conditions cannot be sufficiently handled. If a human operator is part of the system, there may be more exceptions that can be covered, but with less certainty.



ERROR AND EXCEPTION HANDLING AND FAULT TOLERANCE


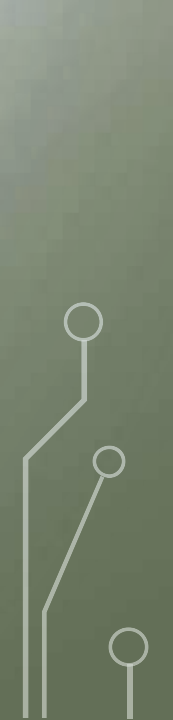
- **Fault tolerance** deals with correcting for known error conditions.
 - Deals with **error handling** in all other cases: ill-designed systems, **faults** in the **exception handling** code, errors originating from outside the system or framework.
 - Enables a system to continue functioning even in the presence of faults.
- 
- 

A decorative graphic on the left side of the slide consisting of a network of thin, light blue lines. These lines form a complex, branching pattern that resembles a circuit board or a stylized tree. Small circles are placed at various points where the lines intersect or terminate, adding to the technical or digital aesthetic of the design.

INTERACTION AND PRESENTATION

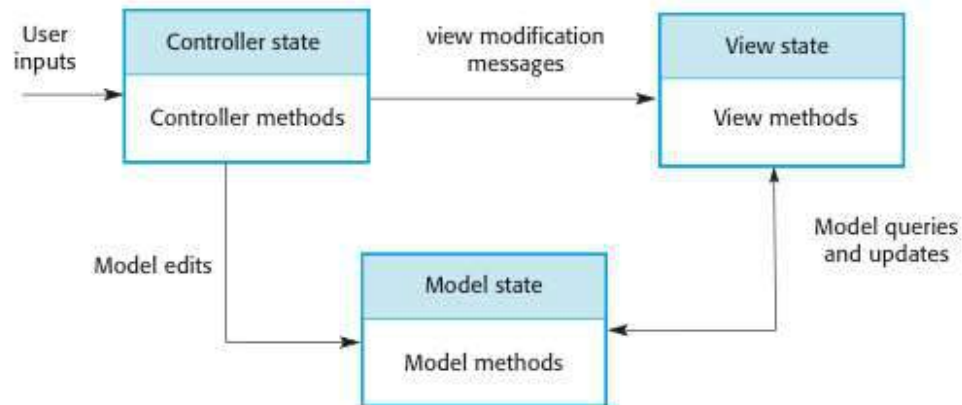


INTERACTION AND PRESENTATION

- This design issue is concerned with how to structure and organize interactions with users as well as the presentation of information.
 - For example, separation of presentation and business logic using the *Model-View-Controller* approach.
- 
- 

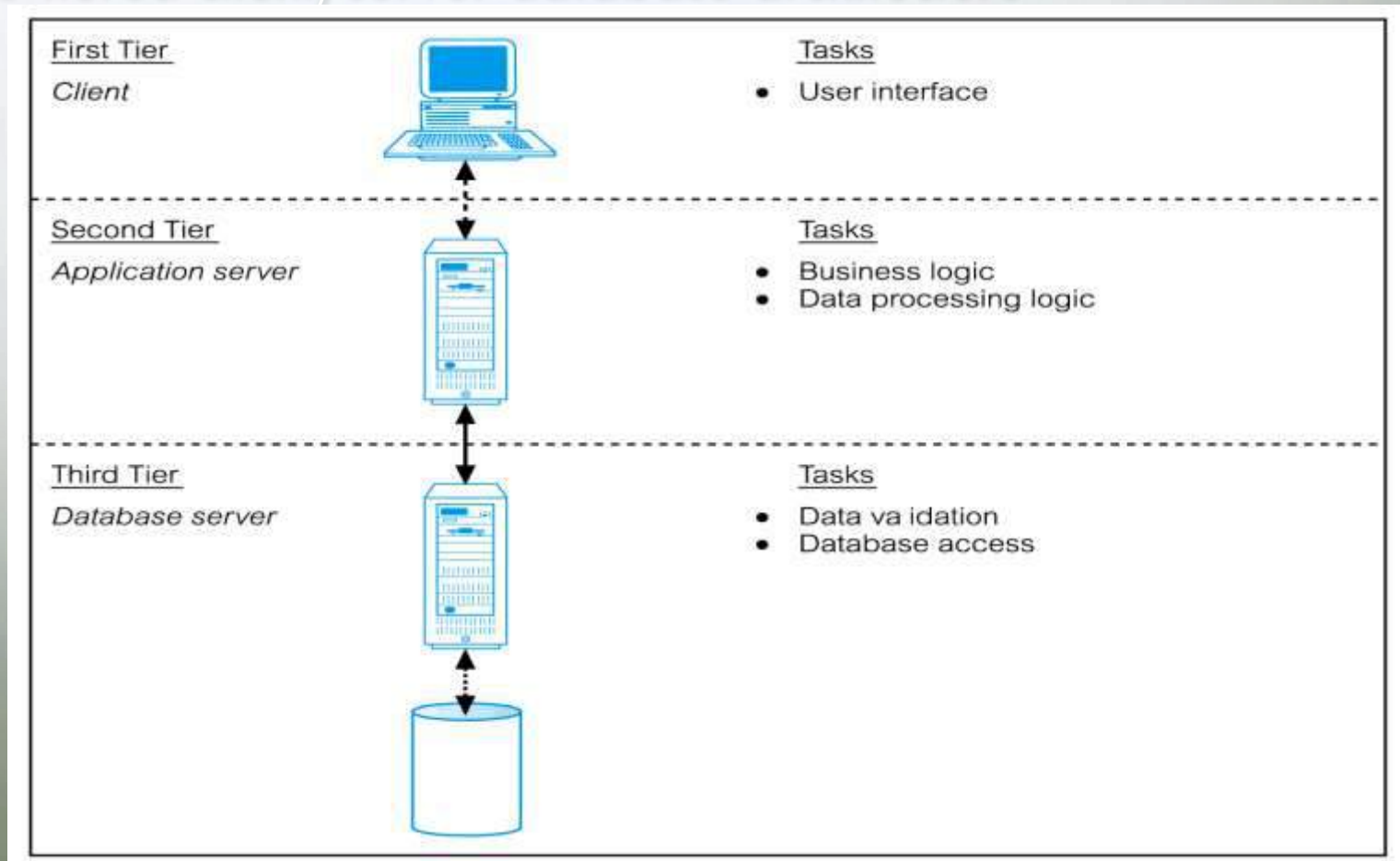
INTERACTION AND PRESENTATION

Figure 16.5 The Model-View-Controller pattern



INTERACTION AND PRESENTATION

- Three-tiered client/server database architecture

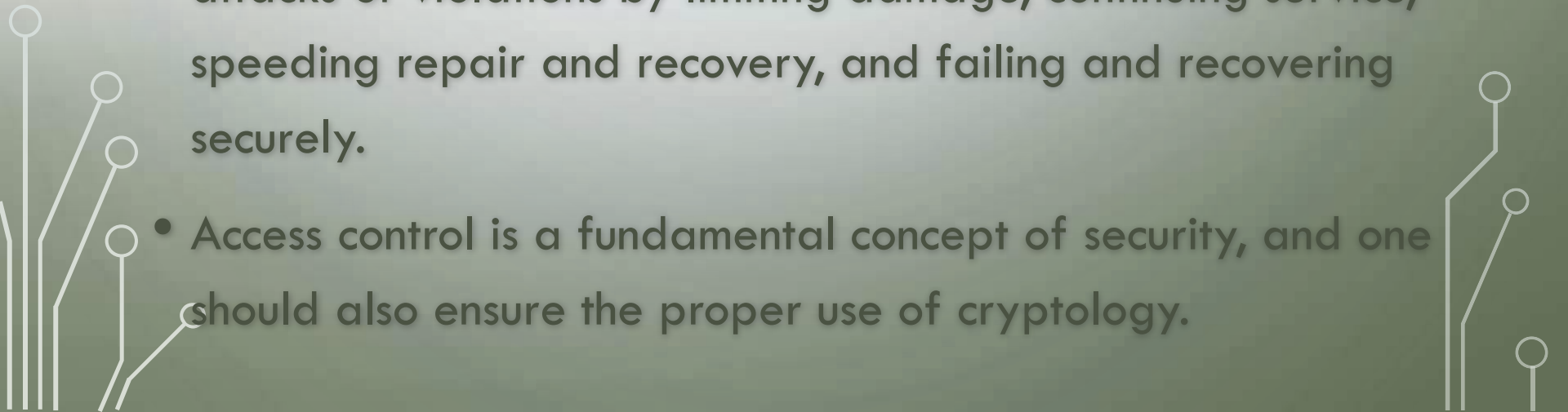




SECURITY


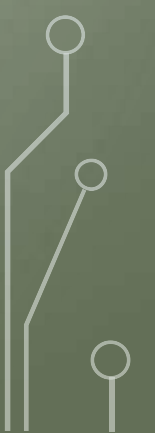


SECURITY

- Design for security is concerned with how to prevent unauthorized disclosure, creation, change, deletion, or denial of access to information and other resources.
 - It is also concerned with how to tolerate security-related attacks or violations by limiting damage, continuing service, speeding repair and recovery, and failing and recovering securely.
 - Access control is a fundamental concept of security, and one should also ensure the proper use of cryptology.
- 


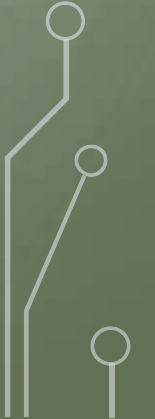


SECURITY

- Security specification has something in common with safety requirements specification – in both cases, your concern is to avoid something bad happening.
 - Types of security:
 - Identification
 - Authentication
 - Authorisation
 - Integrity
 - Etc.
- 
- 


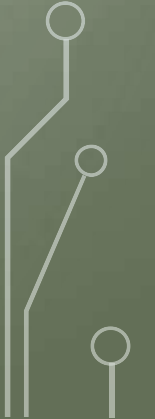


SECURITY

- Four major differences between security and safety:
 - Safety problems are accidental – the software is not operating in a hostile environment. In security, you must assume that attackers have knowledge of system weaknesses
 - When safety failures occur, you can look for the root cause or weakness that led to the failure. When failure results from a deliberate attack, the attacker may conceal the cause of the failure.
- 
- 




SECURITY

- Shutting down a system can avoid a safety-related failure. Causing a shut down may be the aim of an attack.
 - Safety-related events are not generated from an intelligent adversary. An attacker can probe defenses over time to discover weaknesses.
- 
- 



SUMMARY

- This chapter discuss in detail about key issues in software design such as concurrency, control and handling of events, data persistence, distribution of components, error and exception handling and fault tolerance, interaction and presentation and security.
- 
- 