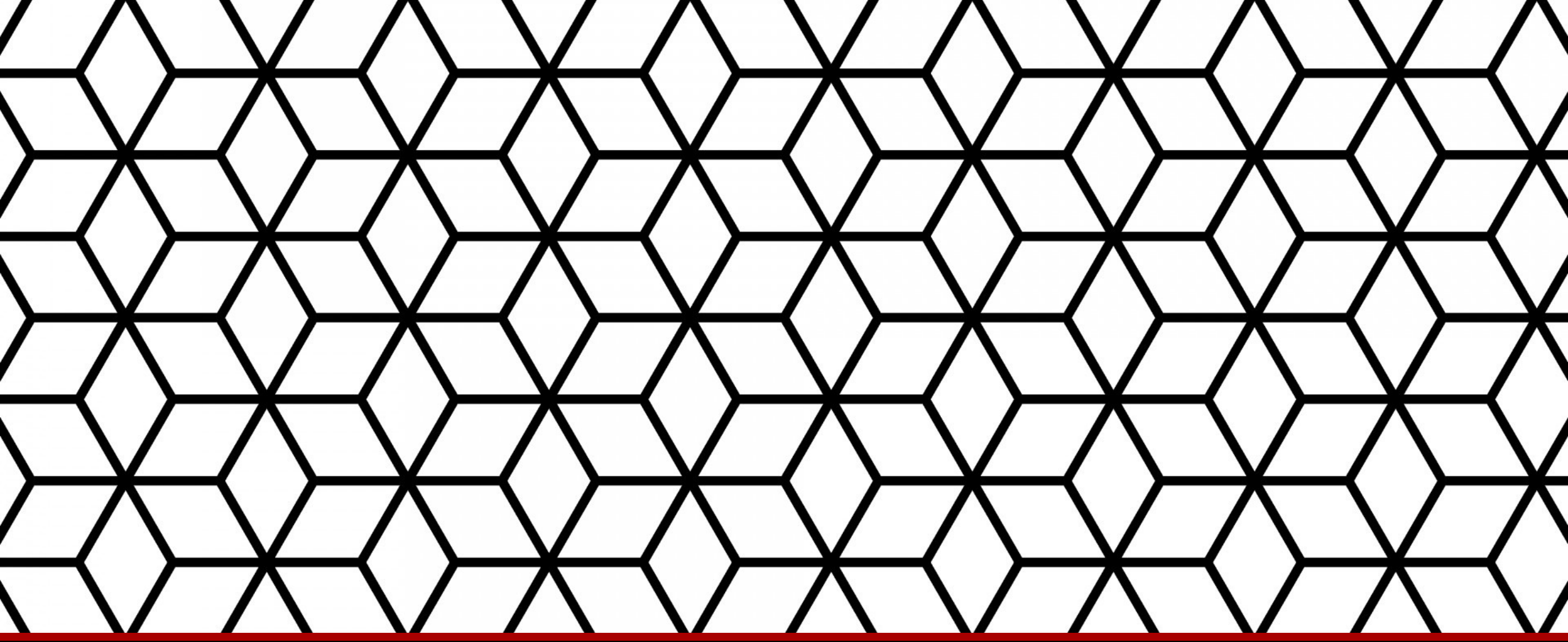
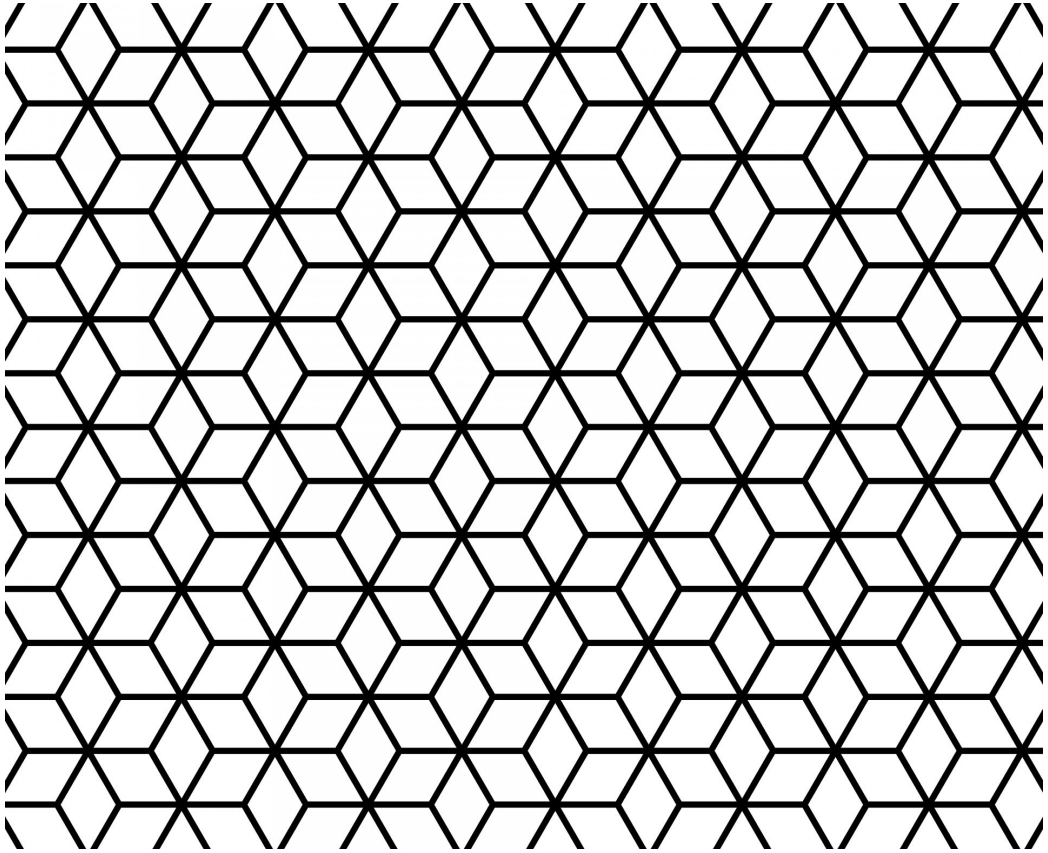


Design Patterns

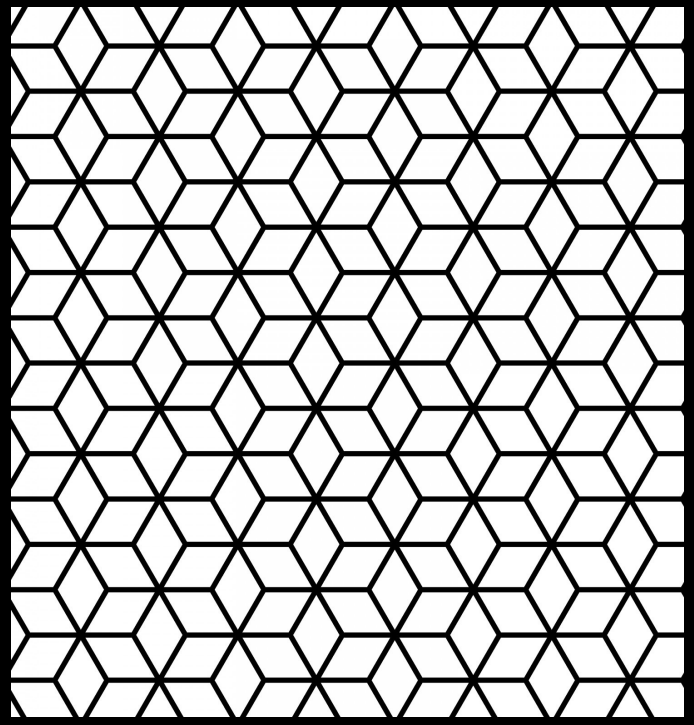
ITMD-510

What is a design pattern?

- A reusable and generalized solution to a problem in software design and architecture
- A proven template or blueprint that can be applied to various situations to solve similar design challenges
- Not complete programs or algorithms; guidelines for creating structures that promote
  - code organization,
  - reusability, and
  - maintainability.



Design Pattern Characteristics



Reusability: Encapsulate reusable solutions reducing the need to reinvent the wheel

Abstraction: Abstract away the implementation details, focusing on the underlying structure and relationships between components

Common Terminology: Establish common vocabulary allowing developers to communicate more effectively about design decisions

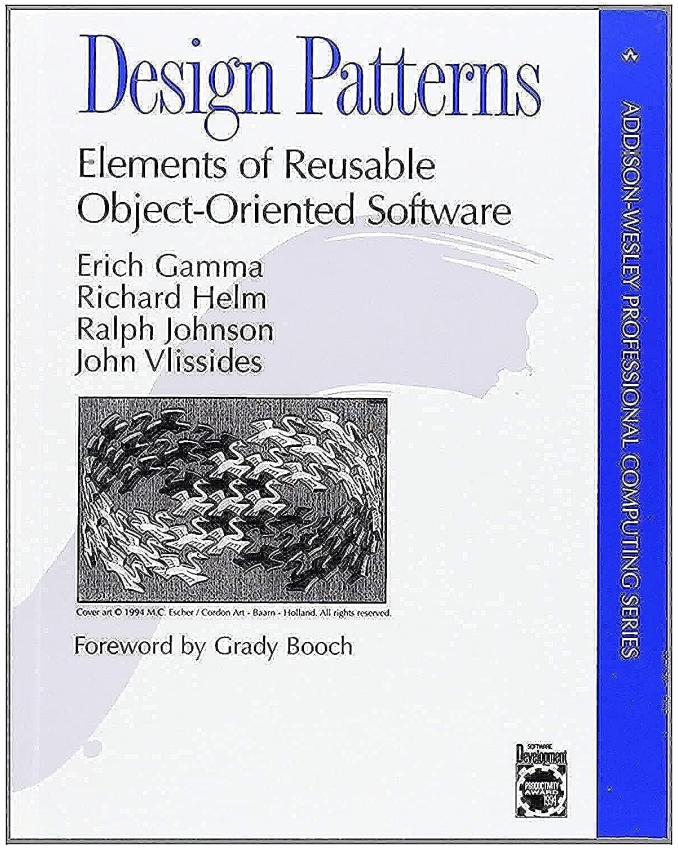
Flexibility: Can be adapted and customized to fit the specific requirements while maintaining the core solution

Best Practices: Embody best practices for solving specific design challenges, contributing to more maintainable and extensible code

Standardization: Promote standardized solutions, making codebases more understandable for other developers

History of design patterns

- In 1994, the software engineering book ***Design Patterns: Elements of Reusable Object-Oriented Software*** was published
- Written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, it explored object-oriented programming, and described patterns for designing software
- The authors became known as the "Gang of Four" (GoF)



Reason for and relevance of the book

- **Lack of Documentation:** Prior to the book, there was a lack of well-documented solutions to recurring design problems
- **Escalating Complexity**
  - Software systems were increasing in complexity which increased the challenges faced by developers
  - Design patterns provided structured, systematic approaches to address these complexities effectively
- **Advent of Object-Oriented Programming (OOP)**
  - The rise of object-oriented programming in the 1980s brought in new challenges
  - Design patterns offered object-oriented, context-specific strategies
- **Collaboration and Communication**
  - Effective software development depends on teamwork and effective communication
  - Design patterns introduced a shared lexicon and a set of conventions for discussing and implementing design decisions, fostering better collaboration

Reasons the book emerged

- **Lack of Documentation:** Prior to the introduction of design patterns, there was a dearth of well-documented solutions to recurring design problems. Experienced developers possessed invaluable knowledge, but it was often locked in their experience and not widely disseminated.
- **Escalating Complexity:** As software systems grew increasingly intricate, developers confronted challenges in managing this escalating complexity. Design patterns emerged as a response to the need for structured, systematic approaches to address these complexities effectively.
- **Advent of Object-Oriented Programming (OOP):** The ascent of object-oriented programming in the 1980s ushered in novel design challenges. Design patterns offered object-oriented, context-specific strategies to tackle these challenges.
- **Collaboration and Communication:** Effective software development often hinges on teamwork and effective communication among team members. Design patterns introduced a shared lexicon and a set of conventions for discussing and implementing design decisions, fostering better collaboration.

Design pattern benefits

- **Reusability and Scalability**: Promote the reuse of proven solutions, saving time and effort in future projects, facilitating seamless scaling
- **Maintainability and Extensibility**
  - Enables a modular approach, making the codebase easier to maintain and extend
  - Changes in one part of the system do not necessarily affect the entire application
- **Performance Optimization**: Certain design patterns optimize system performance by ensuring efficient resource utilization and minimizing redundancy
- **Established Best Practices**
  - Embody best practices derived from industry experience
  - Provide a structured, industry-approved approach to problem-solving

Design pattern categories

- The "Design Patterns" book introduced 23 design patterns
- Since then, developers and researchers have uncovered new patterns and adapted existing ones
- Indispensable tools for development across diverse paradigms and domains, such as web and mobile application development, microservices architecture, etc.
- Various design patterns categories addressing different types of design problems

Creational Patterns: Concerned with object creation, managing object instantiation, encapsulation, and hiding its complexity. Examples include Singleton, Factory Method, and Builder.

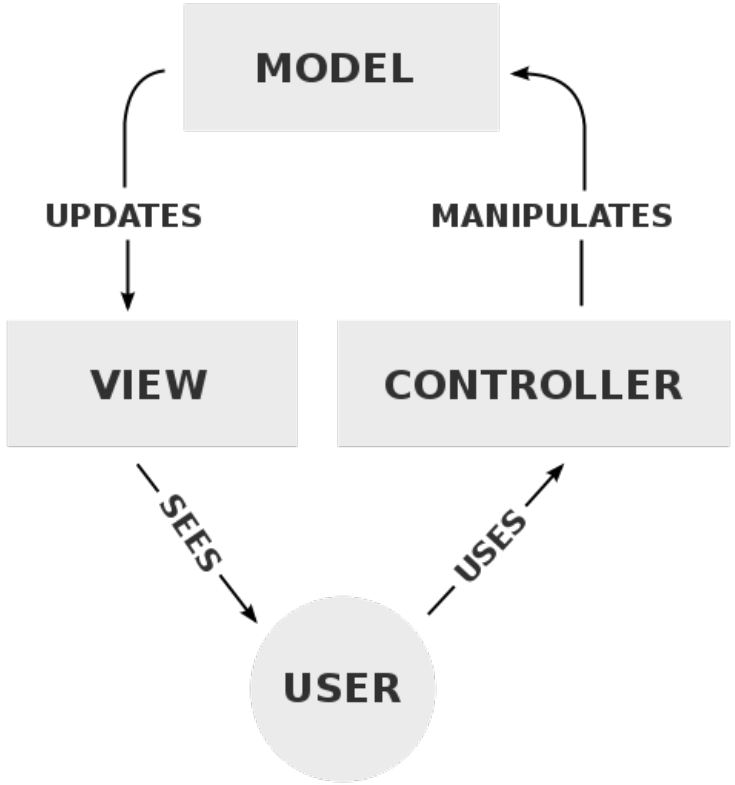
Structural Patterns: Composition of classes or objects to form larger structures, relationships between objects, helping them work together effectively. Examples include Adapter, Bridge, and Composite.

Behavioral Patterns: How objects interact and communicate with one another. Define the flow of control between objects and encapsulate complex control logic. Examples include Observer, Command, and Strategy.

Architectural Patterns: High-level patterns that dictate the overall structure and organization of an application. Examples include **Model-View-Controller (MVC)**, Model-View-ViewModel (MVVM), and Layered Architecture.

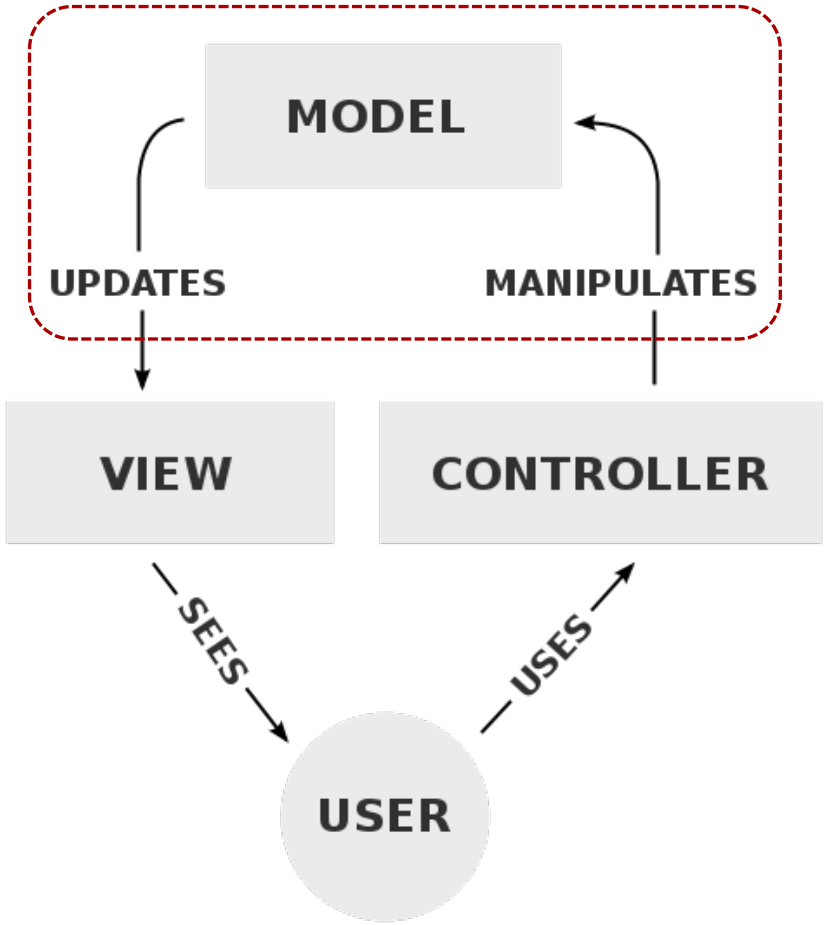
Model-View-Controller (MVC)

- A software architectural pattern used for organizing the structure of applications, **particularly in user interface development**
- Aims to separate the concerns of an application into three main components: the Model, the View, and the Controller
- The separation helps to achieve modularity, maintainability, and ease of development



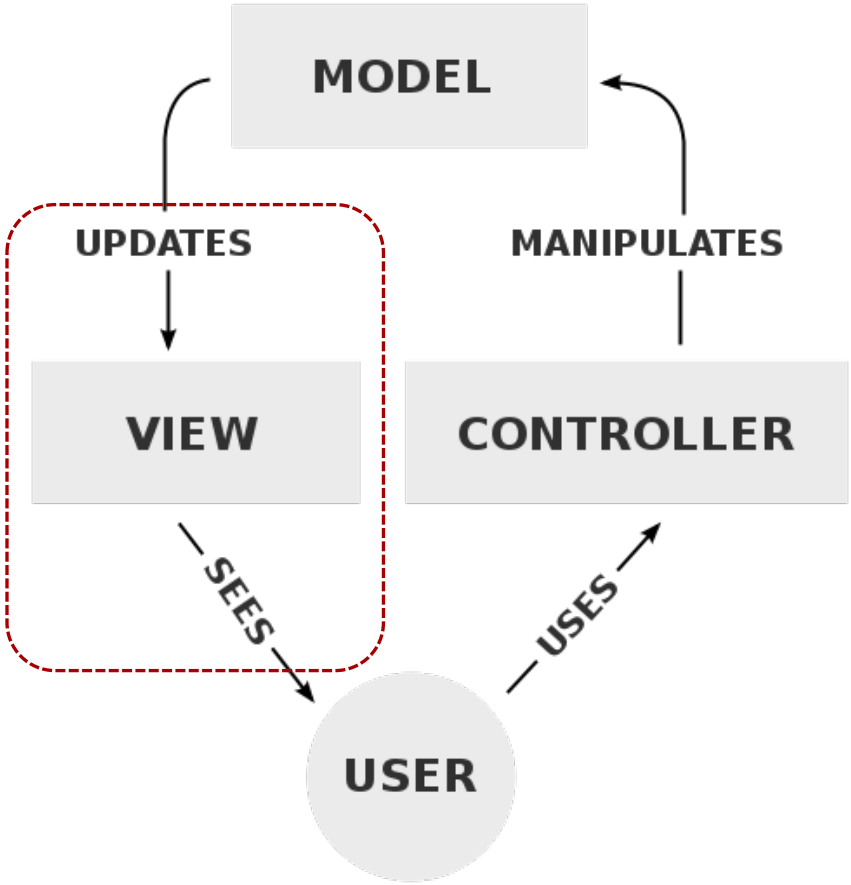
MVC: Model

- Represents the application's data and business logic
- Responsible for managing data, performing computations, and enforcing business rules
- Responds to requests from the Controller
- Notifies the View when there are changes in the data



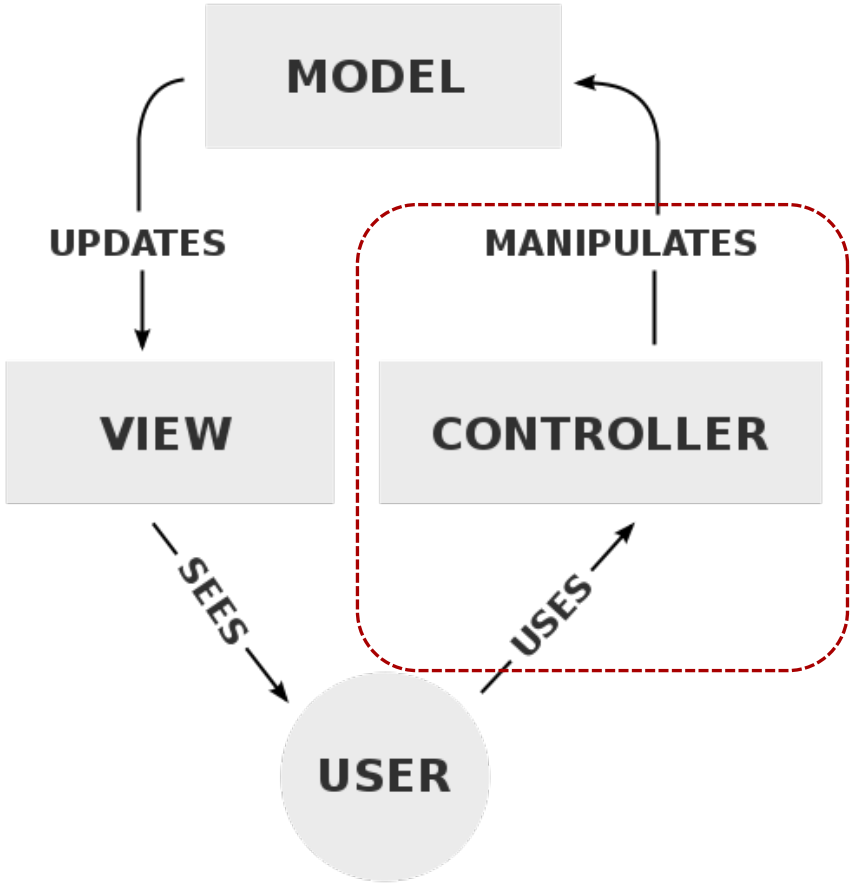
MVC: View

- Responsible for displaying the data provided by the Model to the user
- Handles the presentation layer, including user interfaces, visual elements, and layouts
- Should be passive and ideally have minimal logic
- Mainly focused on presenting data in a human-readable format



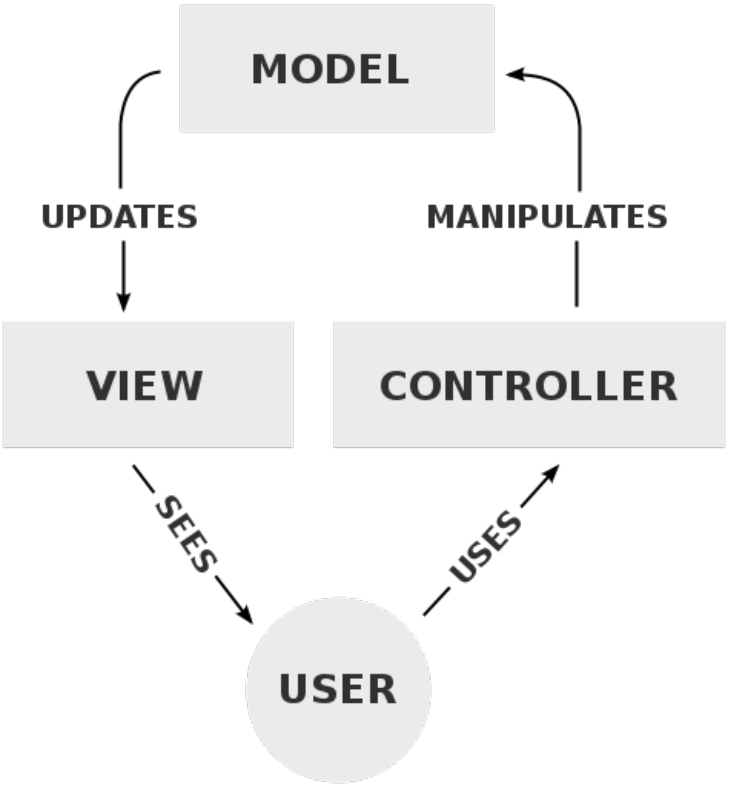
MVC: Controller

- Acts as an intermediary between the Model and the View
- Handles user input, processes requests from the user interface, and updates the Model accordingly
- Also updates the View when the Model's data changes, ensuring that the user interface reflects the current **state** of the application



Benefits of the MVC pattern

- **Modularity**: Each component has a specific responsibility, making it easier to develop, test, and maintain individual parts of the application
- **Separation of Concerns**: The pattern separates data management (Model), user interface (View), and user interaction (Controller), making the application's architecture cleaner and more understandable
- **Reusability**: Since components are loosely coupled, it's possible to reuse Models and Views in different parts of the application or even in different applications
- **Parallel Development**: Different teams or developers can work on different components simultaneously without interfering with each other's work



Revisiting serialization

Serialization refers to the process of converting complex data structures, such as objects or data records, into a format that can be easily stored, transmitted, or reconstructed later

This format is usually a stream of bytes or a text-based representation that can be written to a file, sent over a network, or stored in a database

Serialization is primarily used to...

Persistence

Communication

Persistence

- Save an object's state to disk or a database so that it can be retrieved later
- Commonly used in applications to save user preferences or application state

Communication

- Transmit data between different parts of a program or between different applications over a network
- E.g., when a web server sends data to a web browser, it needs to serialize the data for transmission, and the browser deserializes it

Serialization is primarily used to...

Interoperability

Caching

Interoperability

- When working with multiple programming languages or systems, serialization helps in exchanging data between them
- Data serialization formats like JSON, XML, and Protobuf are used for this purpose

Caching

- In distributed systems, serialized data can be stored in a cache, making it faster to retrieve frequently used information

Common serialization formats and methods

- **JSON (JavaScript Object Notation):** A human-readable and lightweight data interchange format that is easy for both humans and machines to read and write.
- **XML (eXtensible Markup Language):** A text-based format that uses tags to represent structured data. It's commonly used for configuration files and data exchange between different systems.
- **Binary Serialization:** This involves converting data into a binary format, which is more efficient in terms of size and speed but not human-readable
- **Protocol Buffers (Protobuf):** A binary serialization format developed by Google that is efficient and extensible. It is often used in high-performance, distributed systems.
- **MessagePack:** A binary serialization format that is designed for efficiency and speed, often used in data exchange between systems.
- **Custom Serialization:** In some cases, developers may implement custom serialization logic tailored to their specific requirements.

Secure serialization

Working with serialization, you need to consider security aspects, as deserializing data from untrusted sources can lead to security vulnerabilities

This is known as a "serialization attacks" and can be mitigated by

using secure serialization libraries,	validating input data, and	applying appropriate security measures

Serialization in Java

Serialization in Java is the process of converting complex objects or data structures into a format that can be easily stored, transmitted, or reconstructed

This process is essential for persisting objects, transferring data between applications, and supporting distributed systems

Java provides a built-in mechanism for serialization through the **`java.io.Serializable`** interface, and it is commonly used in various scenarios, from saving game states to exchanging data between client and server applications

java.io.Serializable

- The core interface for supporting object serialization
- To make a class serializable, you need to implement this interface

```
import java.io.Serializable;

public class Person implements Serializable {
    // Class members and methods here
}
```

- By implementing Serializable, you indicate that an object of this class can be converted into a stream of bytes and reconstructed later

Serializing Objects

Use the **`ObjectOutputStream`** class to serialize objects, which writes the object's state to an output stream.

```
import java.io.*;

public class SerializationExample {
    public static void main(String[] args) {
        try {
            Person person = new Person("John Doe", 30);
            ObjectOutputStream out =
              new ObjectOutputStream(new FileOutputStream("person.ser"));
            out.writeObject(person);
            out.close();
            System.out.println("Object has been serialized.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, we create a **`Person`** object and serialize it to a file named "**`person.ser`**". The **`ObjectOutputStream`** takes care of writing the object's state to the file.

Deserializing Objects

To deserialize an object, you can use the **`ObjectInputStream`** class, which reads an object's state from an input stream. Here's how to do it:

```
public class DeserializationExample {
    public static void main(String[] args) {
        try {
            ObjectInputStream in =
               new ObjectInputStream(new FileInputStream("person.ser"));
            Person person = (Person) in.readObject();
            in.close();
            System.out.println("Object has been deserialized.");
            System.out.println("Name: " + person.getName());
            System.out.println("Age: " + person.getAge());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, we read the serialized **`Person`** object from "**`person.ser`**" and cast it back to a **`Person`** object.

Summary

- Serialization is a fundamental concept in Java, providing a means to store and exchange object data efficiently
- You can easily serialize and deserialize objects by
  - implementing the **`java.io.Serializable`** interface and
  - utilizing the **`ObjectOutputStream`** and **`ObjectInputStream`** classes
- Custom serialization allows for greater control when needed
- Understanding serialization is crucial for various Java applications, and it opens the door to many advanced use cases and scenarios

Supplemental References & Resources

Gamma , E., Helm, R., Johnson, R., & Vlissides , J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

GeeksforGeeks. (2018, August 31). Software Design Patterns. Retrieved from GeeksforGeeks.com: https://www.geeksforgeeks.org/software-design-patterns/

Karimov, I. (2020, December 18). Design Patterns. Retrieved from Medium: https://medium.com/nerd-for-tech/design-patterns-f6ee70d13296

Oracle Corporation . (n.d.). Java Object Serialization. Retrieved from Java Object Serialization - docs.oracle.com: https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html

Oracle Corporation. (n.d.). Object Serialization Examples. Retrieved from Object Serialization Examples - docs.oracle.com: https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/examples/index.html

Sarcar, V. (2022). Java Design Patterns: A Hands-On Experience with Real-World Examples. Apress.

tutorialspoint. (n.d.). Design Pattern - Overview. Retrieved from tutorialspoint: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm

Wengner, M. (2023). Practical Design Patterns for Java Developers: Hone your software design skills by implementing popular design patterns in Java. Packt Publishing.