UML and Design Patterns

Object-Oriented Programming

Introduction to UML and Patterns

- Software design tools applicable across programming languages
- Require object-oriented programming (OOP) features
- UML (Unified Modeling Language)
  - Graphical language for software design and documentation
  - Used within the OOP framework
- What are Design Patterns?
  - Template or outline for software tasks
  - Can be implemented as different code in similar applications
- Benefits of UML and Patterns
  - Enhance software design process
  - Improve code reusability
  - Facilitate communication among developers
  - Promote best practices in OOP

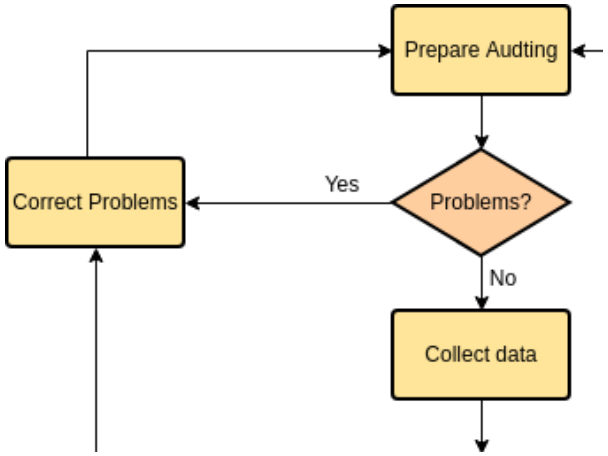
Human-Oriented Representations

- Needs
  - Most people don't think in programming languages
  - Computer scientists seek more intuitive ways to represent programs
  - Pseudocode: mixture of programming and natural language
- Limitations of Pseudocode
  - Standard tool for programmers
  - Linear and algebraic representation
  - Lacks graphical elements

Evolution of Graphical Representations

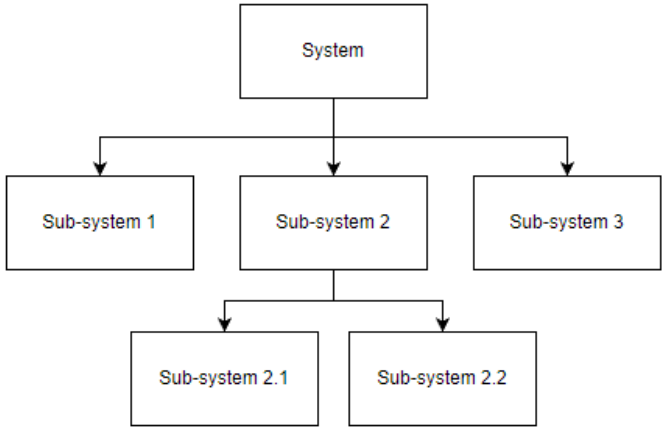
- Past attempts: flowcharts, structure diagrams
- Many graphical representations now outdated
- UML: Current candidate for graphical representation

flowcharts



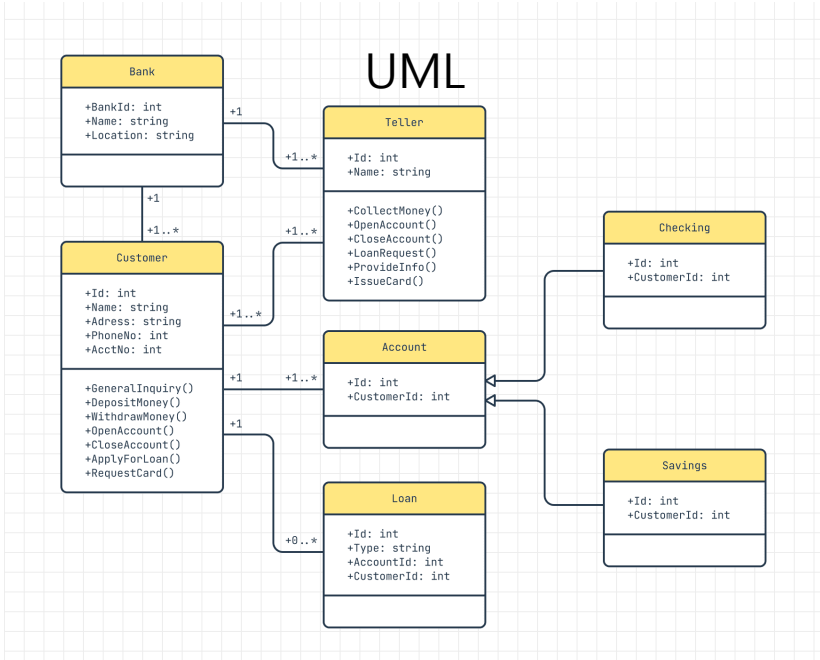
online.visual-paradigm.com

structure diagrams



https://www.savemyexams.com/

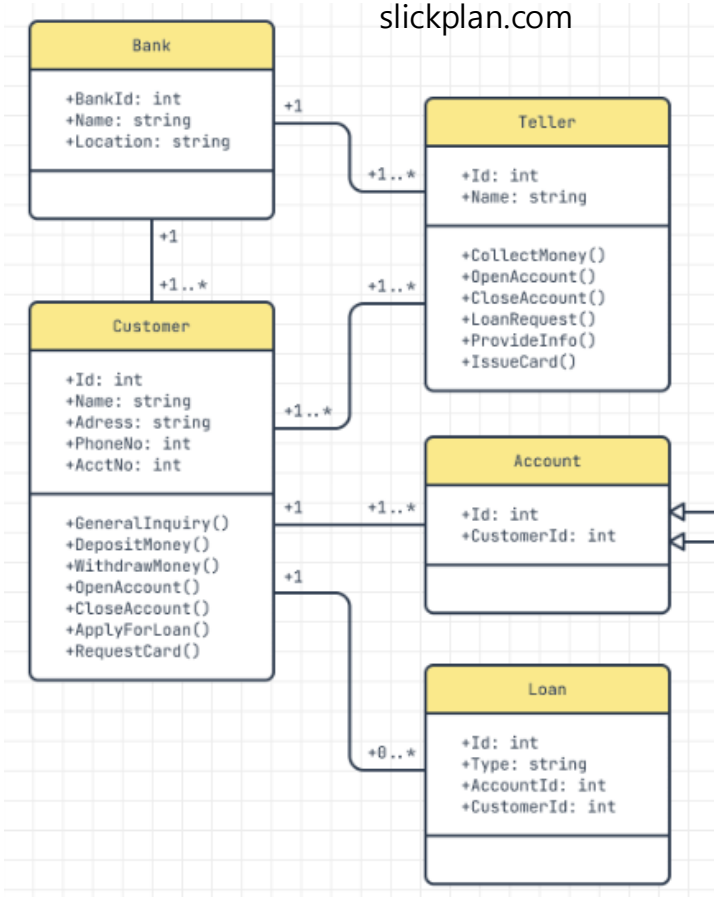
UML



slickplan.com

UML and Object-Oriented Programming

- UML: Designed to reflect OOP philosophy
- Gaining adoption in software design projects
- Still evolving and being tested

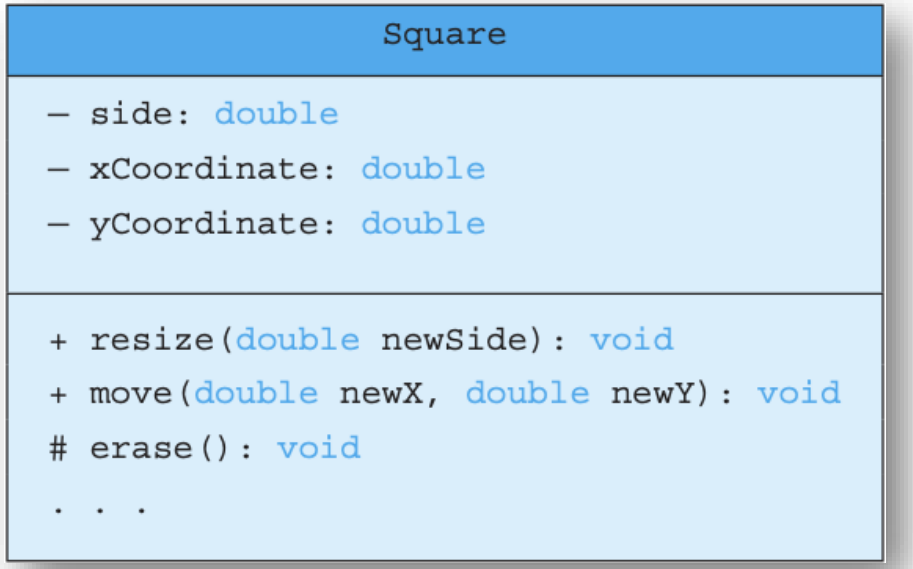


History of UML

- UML and OOP
  - UML developed alongside Object-Oriented Programming (OOP)
  - Various groups created their own representations for OOP design
- Birth of UML (1996)
  - Created by Grady Booch, Ivar Jacobson, and James Rumbaugh
  - Goal: Standardize graphical representation for OO design and documentation
- UML Today
  - Maintained and certified by Object Management Group (OMG)
  - OMG: Nonprofit organization promoting object-oriented techniques

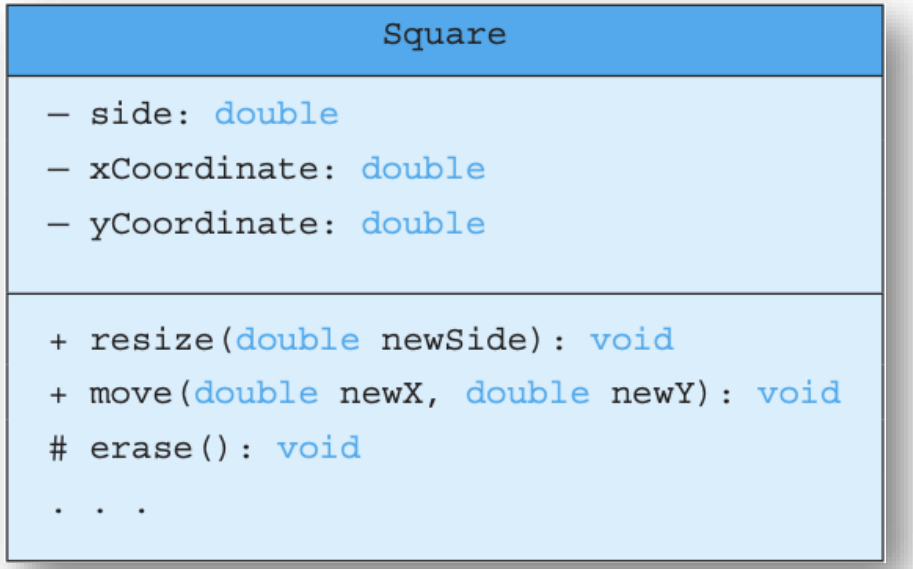
UML Class Diagrams (1/2)

- UML Class Diagrams
  - Central to Object-Oriented Programming (OOP)
  - Easy to understand and use
  - Represents a class structure graphically
- Structure of a Class Diagram
  - Box divided into three sections
    - Class name
    - Data specification (instance variables)
    - Actions (class methods)
  - Optional color coding (not standardized)



UML Class Diagrams (2/2)

- Access Modifiers in Class Diagrams
  - Minus sign (-): private member
  - Plus sign (+): public member
  - Sharp (#): protected member
  - Tilde (~): package access
- Incomplete Class Diagrams
  - Not all members need to be listed
  - Ellipsis (...) indicates missing members
  - Useful for focused analysis

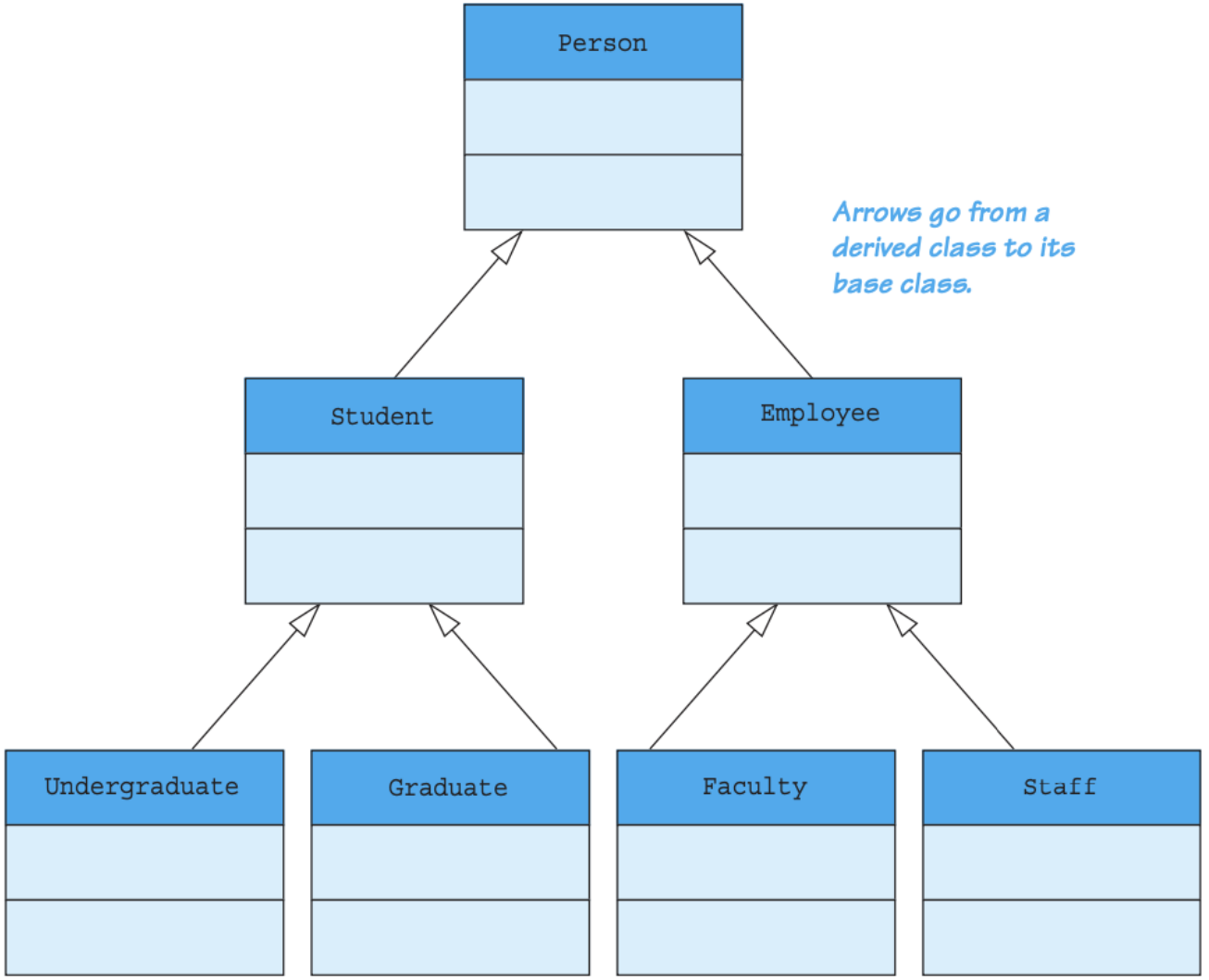


Class Interactions in UML

- Class diagrams alone have limited value
- UML provides ways to show class interactions:
  - Annotated arrows for information flow
  - Package groupings
  - Inheritance annotations
  - Other interaction annotations
- Extensibility of UML
  - UML can be extended for specific needs
  - Extensions follow a prescribed framework
  - Ensures understanding among different developers

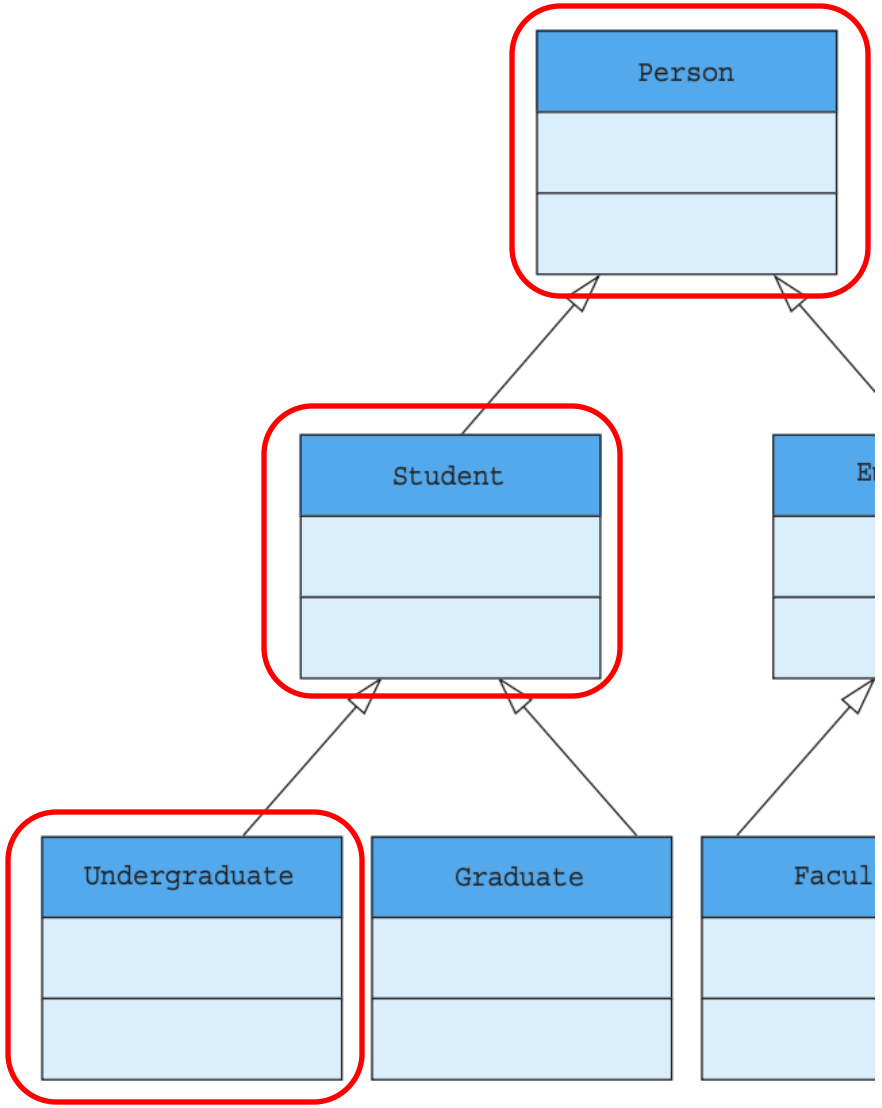
Inheritance Diagrams in UML

- Used to represent class hierarchies
- Example: University record-keeping software
- Key Features of Inheritance Diagrams
  - Arrows point from derived (child) class to base (parent) class
  - Unfilled arrowheads indicate inheritance relationship



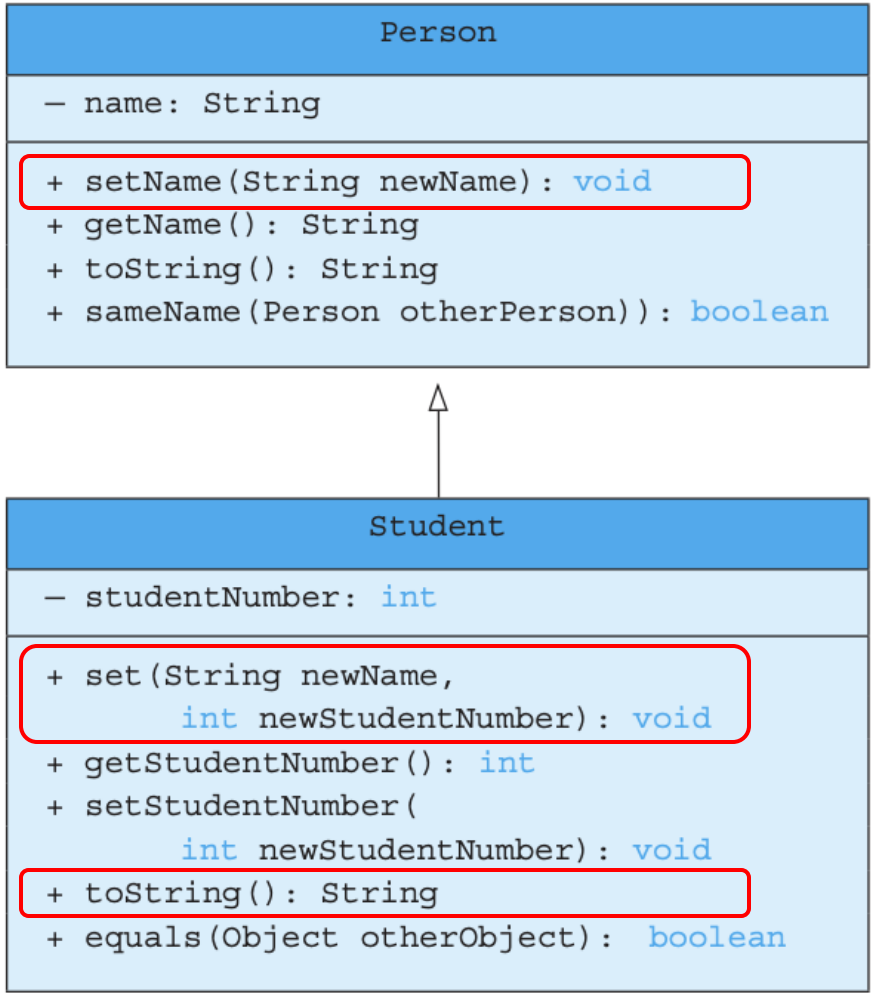
Method Definition Location

- Arrows guide method definition search
- Example: Undergraduate class method
  - Look in Undergraduate class
  - If not found, look in Student class
  - If still not found, look in Person class



Detailed Example for UML Class Hierarchy

- Student s = new Student(); 일 때
- s.toString()과 s.set("Joe", 4242) 는 class Student에서 찾을 수 있음
- s.setName("Josephine") 은 Student에 없기 때문에 자신의 parent로 가서 Person class에서 찾을 수 있음



Design Patterns

- Design outlines applicable across various software applications
- Must be useful across different situations
- Make assumptions about application domains
- Container-Iterator Pattern
  - Container: Class holding multiple data pieces (e.g., array, vector, linked list)
  - Iterator: Construct to cycle through container items
  - Example: Array index as iterator (iterator ‘i’)

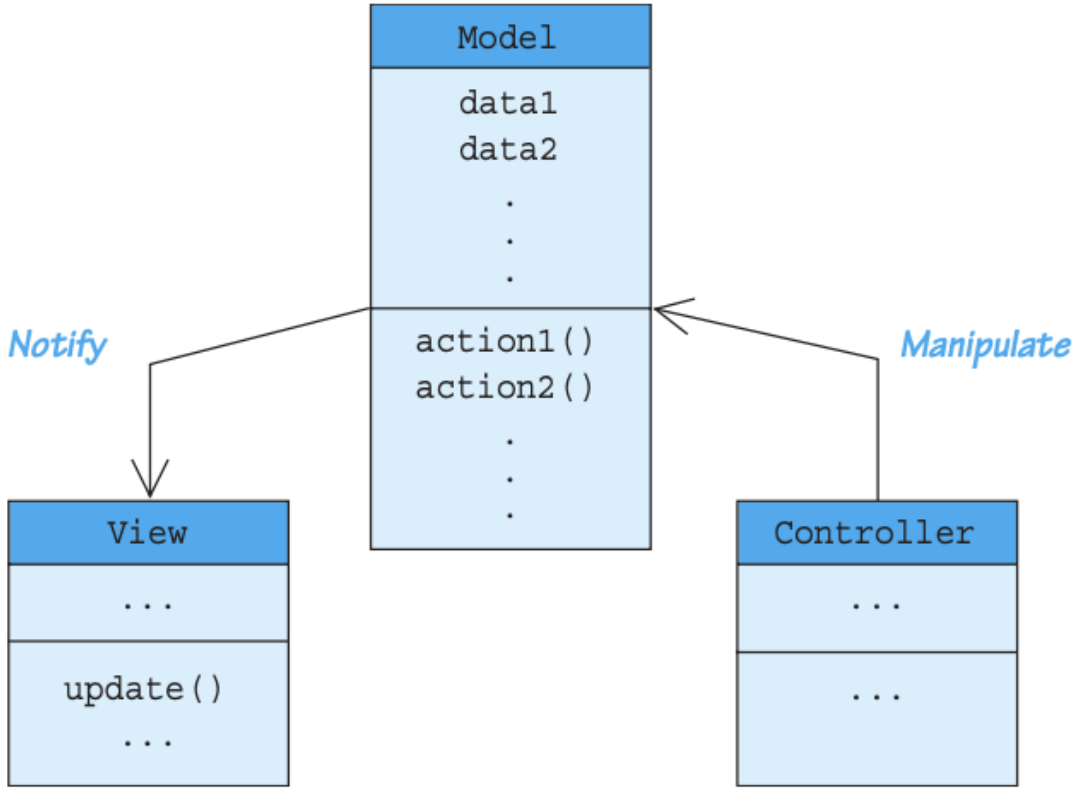
```
for (int i; i < a.length; i++)
   Do something with a[i]
```

Adaptor Pattern

- Transforms one class into a different class
- Adds new interface without changing underlying class
- Example
  - Creating a stack from an array
  - Creating a queue from a linked list

Model-View-Controller (MVC) Pattern

- Separates I/O tasks from the rest of the application
- Model: Core functionality
- View: Output display
- Controller: Input handling



MVC Pattern Example

- Model: Container class (e.g., array)
- View: Display of array element
- Controller: Commands to display specific index
- Suitable for GUI design projects

Efficient Sorting Pattern

- Common pattern in efficient sorting algorithms
- Recursive approach
- Divide, sort, and recombine strategy

Divide-and-Conquer Sorting Pattern

```
/**
Precondition: Interval a[begin] through a[end] of a have elements.
Postcondition: The values in the interval have
been rearranged so that a[begin] <= a[begin+1] <= . . . <= a[end].
*/
public static void sort(Type[] a, int begin, int end) {
   if ((end - begin) >= 1) {
      int splitPoint = split(a, begin, end);
      sort(a, begin, splitPoint);
      sort(a, splitPoint+1, end);
      join(a, begin, splitPoint, end);
   }
   else {
            //else sorting one (or fewer) elements so do nothing.
   }
}
```

core of sorting pattern

Split and Join Methods

- Split: Rearranges and divides the array interval
- Join: Combines two sorted intervals
- Different implementations lead to different sorting algorithms
- Flexibility of the Pattern
  - Split method can be implemented in various ways
  - Simple division or more elaborate rearrangement
  - Adaptable to different sorting strategies

Future of Design Patterns

- Evolving field in software engineering
- Many known patterns, more to be discovered
- Continuous development and refinement