Design Patterns

CS6.401 Software Engineering

Dr. Karthik Vaidhyanthan

karthik.vaidhyanathan@iiit.ac.in

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF
INFORMATION TECHNOLOGY
HYDERABAD

SERC

Software Engineering Research Centre

Acknowledgements

The materials used in this presentation have been gathered/adapted/generated from various sources as well as based on my own experiences and knowledge

-- Karthik Vaidhyanathan

Sources:

1. **Design Patterns: Elements of Reusable Object-Oriented Software** by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
2. **Head first Design Patterns**, Second Edition, Eric Freeman and Elisabeth Robson

We can always use an adapter: Adapter Pattern! [Structural]

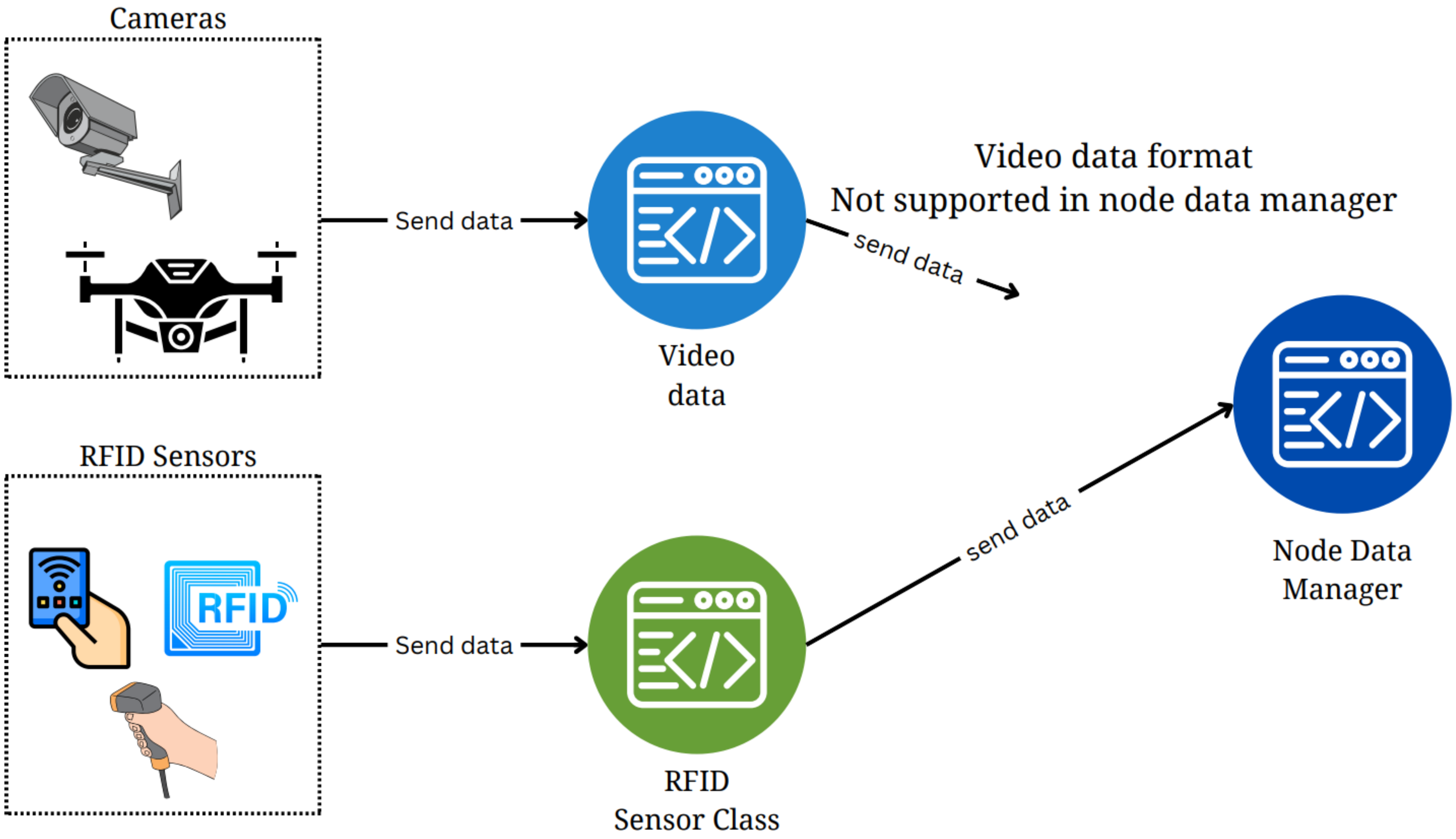
Meet the Adapter Pattern!

Indian

European

Universal adapter

Meet the Adapter Pattern – A Scenario



Why don’t we write an adapter that can transform?

Meet the Adapter Pattern

- What if the interfaces are incompatible?
- What if we can have an adapter in between that can transform the new format?
- Adapter wraps the complexity of conversion
- Supports collaboration of different types of object
- Two-way adapter can also be made

Adapter Pattern: Documentation

Intent

Convert the interface of a class into another interface expected by the clients

Also Known As: Wrapper

Motivation

- Not every time there are compatible interfaces
- Promote reusability
- Three key objects: *Client, Target, Adapter*

Example: Adapter to transform data [Think of legacy class that accepts only certain formats]

Adapter Pattern: Documentation

Applicability

- There is an existing class but its interface does not match the one needed
- Creation of reusable class that can work with unforeseen classes
- There are several existing subclasses but impractical to adapt their interface by subclassing everyone
  - Use object adapter [The one we use here] – Uses composition
  - Class adapter relies on multiple inheritance

Adapter Pattern: Documentation

Structure

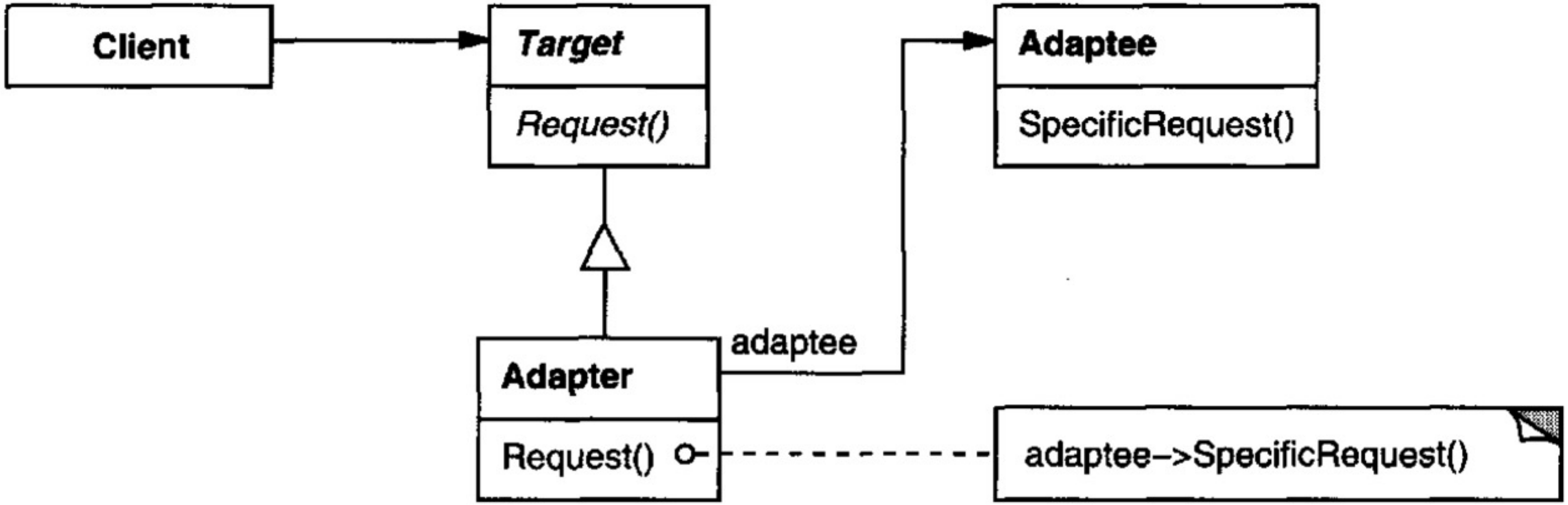


Image source: Gang of four book

Adapter Pattern: Documentation

Participants

Target (NodeData)

- Defines the domain specific interfaces that the client uses

Client (NodeManager)

- Collaborates with objects conforming to their target interfaces

Adaptee (VideoNode)

- Defines an existing interface that needs adapting

Adapter (VideoNodeAdapter)

- Adapts the interface of the Adaptee to the Target interface

Adapter Pattern: Documentation

Consequences

- Single adapter can be used for many adapteees
  - Can implement different functionalities to work with many adaptees
  - New types of adapter can also be easily introduced
- Provides good separation of concerns
  - Keep the logic for conversion in one
  - No need to change at multiple places
- Overall complexity may increase – How much of adaptation is done?
  - Can it be done in a simpler manner on the Adaptee or Target?

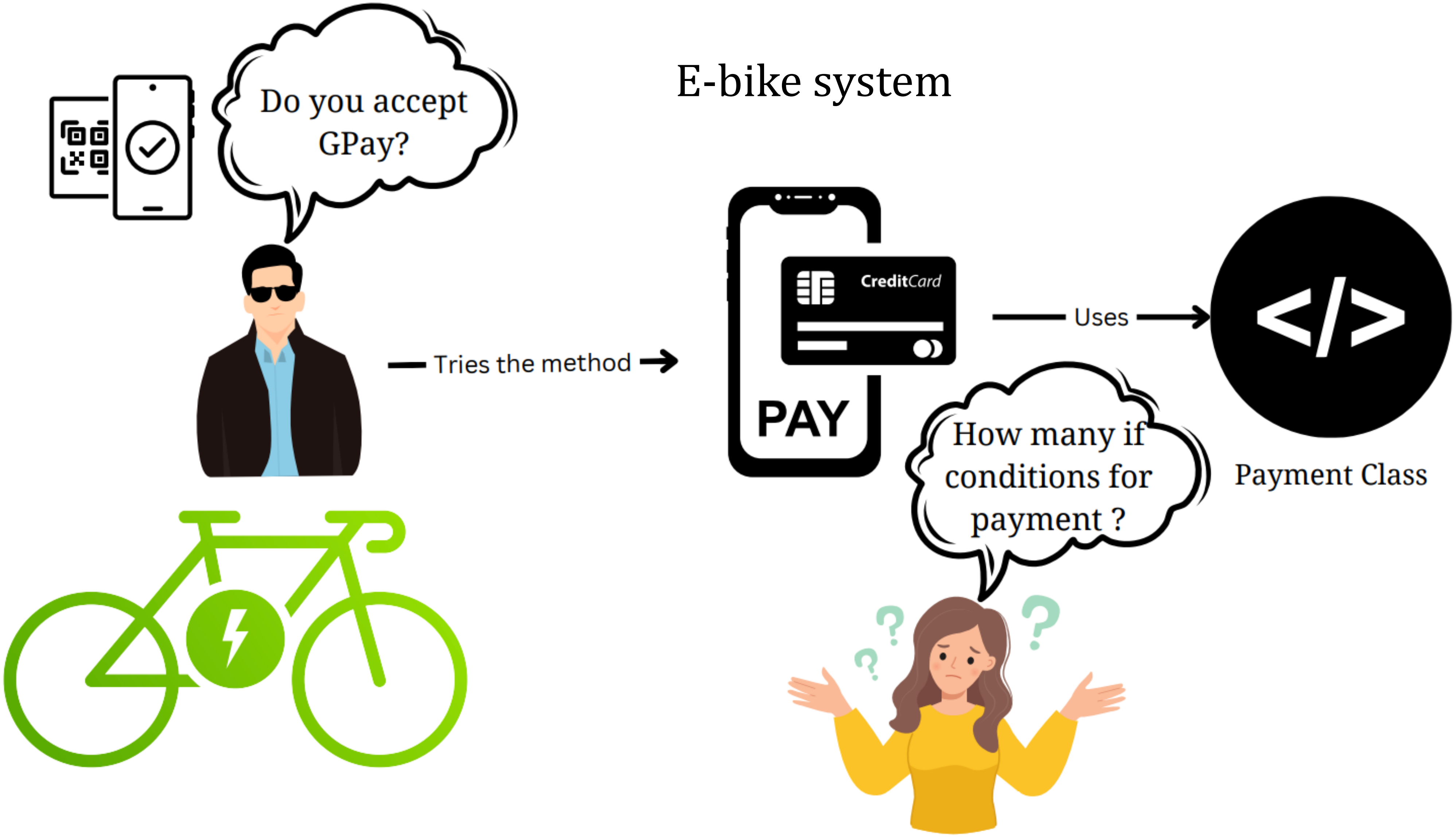
Adapter Pattern: Documentation

Implementation

Check the source code given along: IoTAdapter

Strategies can be different: Strategy Pattern! [Behavioral]

Meet the Strategy Pattern!



E-bike system

Do you accept GPay?

Tries the method

Uses

Payment Class

How many if conditions for payment ?

Meet the Strategy Pattern

- What if you want to alter objects behavior at run-time?
- What if there are similar objects but the way they work is different?
- Each variety of algorithm may require its own set of data and functions

Strategy Pattern: Documentation

Intent

Define a family of algorithms, encapsulate each one and ensure they are interchangeable. Strategy lets algorithm change depending on the client, who is using it

Also Known As: Policy

Motivation

- Different algorithms will be appropriate at different times
- Promotes maintainability
- Two key objects: *Context and Strategy*

Example: Think of Google maps -> selection of mode of transport

Strategy Pattern: Documentation

Applicability

- Many related classes differ only in their behavior
- There is a need for different variants of an algorithm
- Algorithm might require data that client needs not know about – avoid exposing algorithm specific data structures
- Class defines many behaviors and these appear as multiple conditional statements

Strategy Pattern: Documentation

Structure

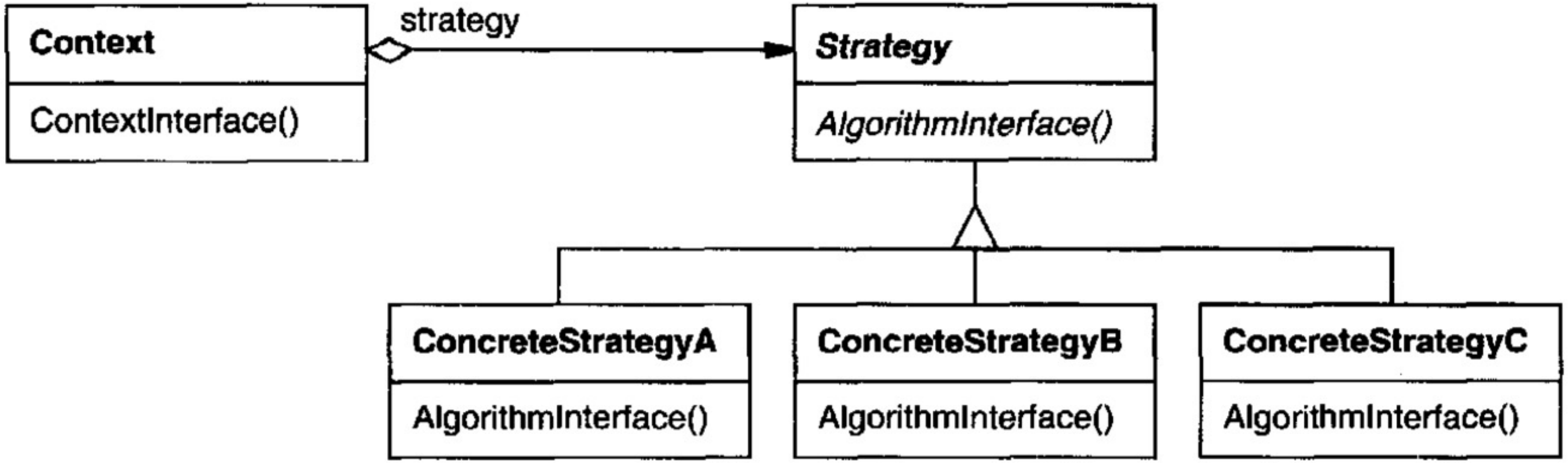


Image source: Gang of four book

Adapter Pattern: Documentation

Participants

Strategy

- Interface common to all algorithms. Used by context

ConcreteStrategy

- Implements algorithm using strategy interface

Context

- Configured with ConcreteStrategy object
- Maintains reference to a Strategy object
- Can define interface for Strategy to access data

Strategy Pattern: Documentation

Consequences

- Families of related algorithms
  - Hierarchies of strategy classes define a family of algorithms or behvaiors
  - Inheritance can help in factoring out common functionality

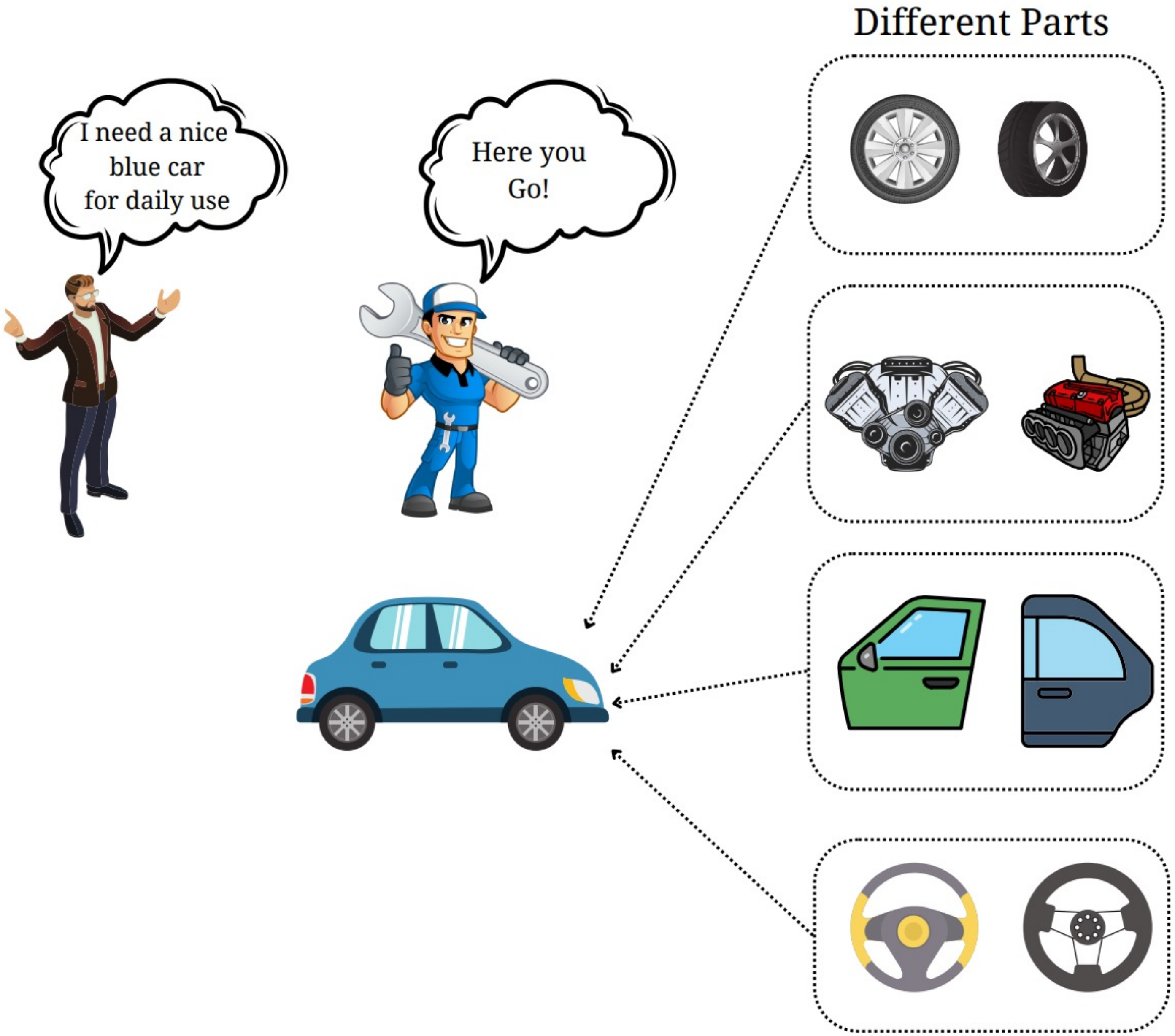
- Alternative to subclassing
  - Inheritance is another mechanism – Hard-wires context [coupling!]

- Eliminates conditional statements
  - Encapsulates behavior separately [Good solution for long method smell]

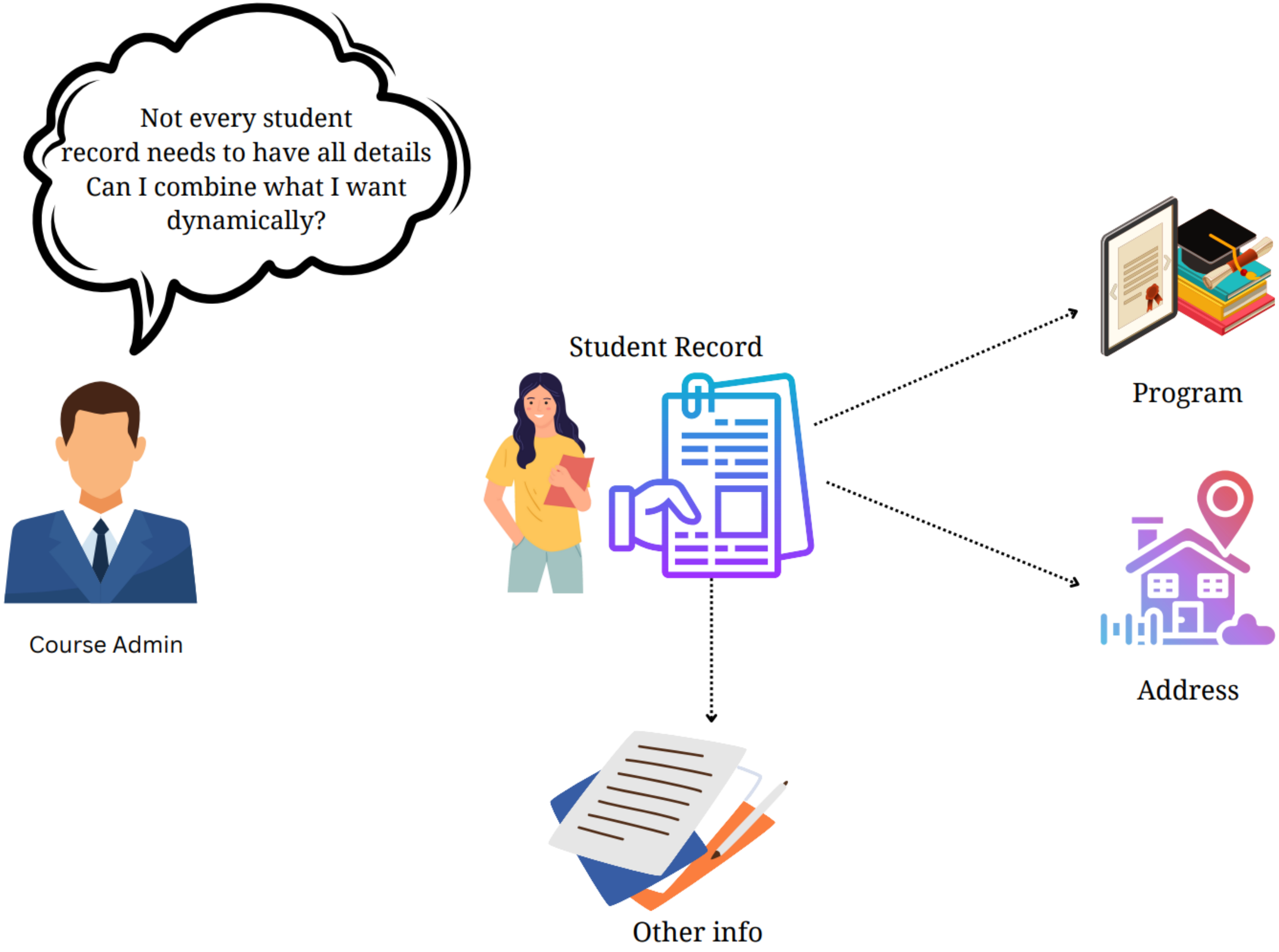
- If the number of variations are less - Don’t overcomplicate!
- Classes must be aware of different possible strategies

How about building things: Builder Pattern! [Creational]

Meet the Builder Pattern!



Meet the Builder Pattern!



Not every student record needs to have all details Can I combine what I want dynamically?

Course Admin

Student Record

Program

Address

Other info

How to dynamically build the different types of student records?

Meet the Builder Pattern

- What if there is a complex object?
- Can we avoid instantiation of a huge constructor?
- Not every time all constructor parameters are required
- Allows extraction of object construction code to separate object
- Creation of an object is just about assembling other objects step by step
- A very decoupled approach to creation

Builder Pattern: Documentation

Intent

Separate construction of complex object from representation such that same construction process can result in different representations

Also Known As: Builder

Motivation

- Separate object construction from business logic
- Promote readability and understandability
- Three key objects: *Director, Builder, Product*

Example: Builder to build different types of vehicles [Each has engine, tyre, etc]

Builder Pattern: Documentation

Applicability

- Algorithm for creating the object must be independent
  - Different parts may make up the object
  - Need not worry about how they are put together

- Construction of different representations of the object needs to be supported

Builder Pattern: Documentation

Structure

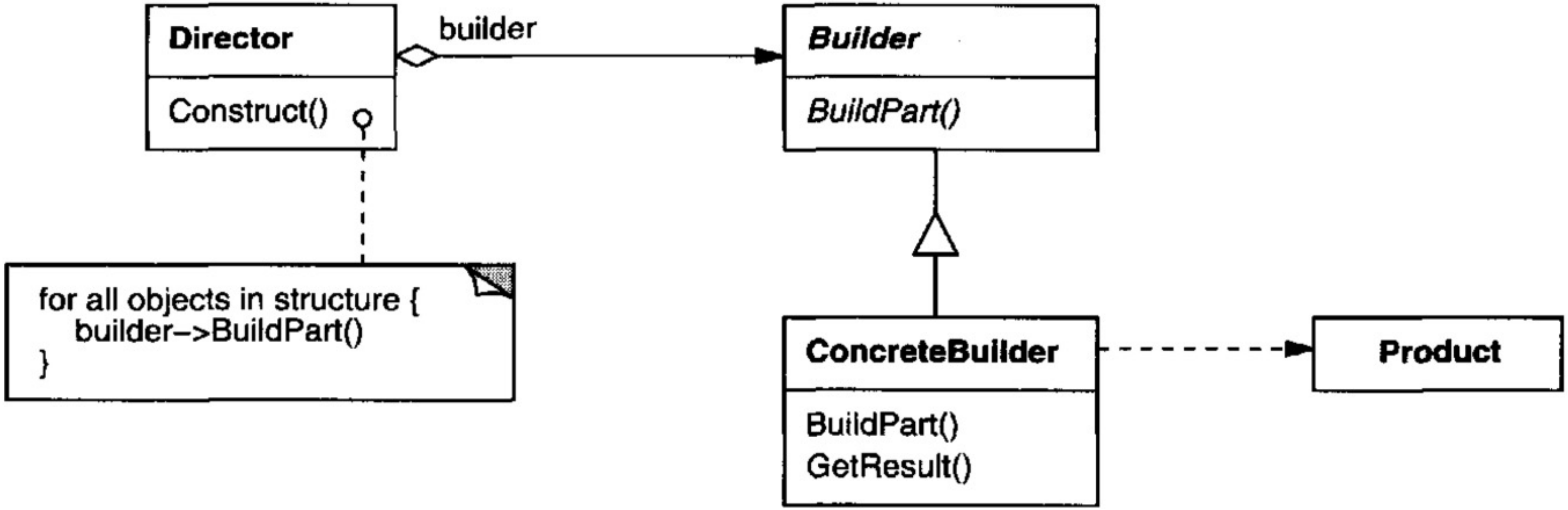


Image source: Gang of four book

Adapter Pattern: Documentation

Participants

Builder (StudentBuilder)

- Defines the interface for creating parts of a product object

ConcreteBuilder (ConcreteStudentBuilder)

- Assembles the parts to create product by implementing builder interface

Director (StudentDirector)

- Constructs an object using the builder interface

Product (Student)

- Complex object under construction
- Includes classes that define the different parts

Builder Pattern: Documentation

Consequences

- Easily vary products internal representation
  - Director gets the abstract interface to build a product
  - All that needs to be done is to define a new kind of builder

- Isolate code for representation and constructions
  - Concrete builder contains code for building a kind of product
  - Directors can reuse builders to build different variants of product

- More control over the construction process
  - Step by step approach under directors control – Focus is on the process

- The overall code complexity increases due to multiple classes
  - Benefits in the long run

Builder Pattern: Documentation

Implementation

Check the source code given along: StudentRecordBuilder

Thank You

Course website: karthikv1392.github.io/cs6401_se

Email: karthik.vaidhyanathan@iiit.ac.in
Web: https://karthikvaidhyanathan.com
Twitter: @karthi_ishere