Design Smells

CS6.401 Software Engineering

Dr. Karthik Vaidhyanthan

karthik.vaidhyanathan@iiit.ac.in

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF
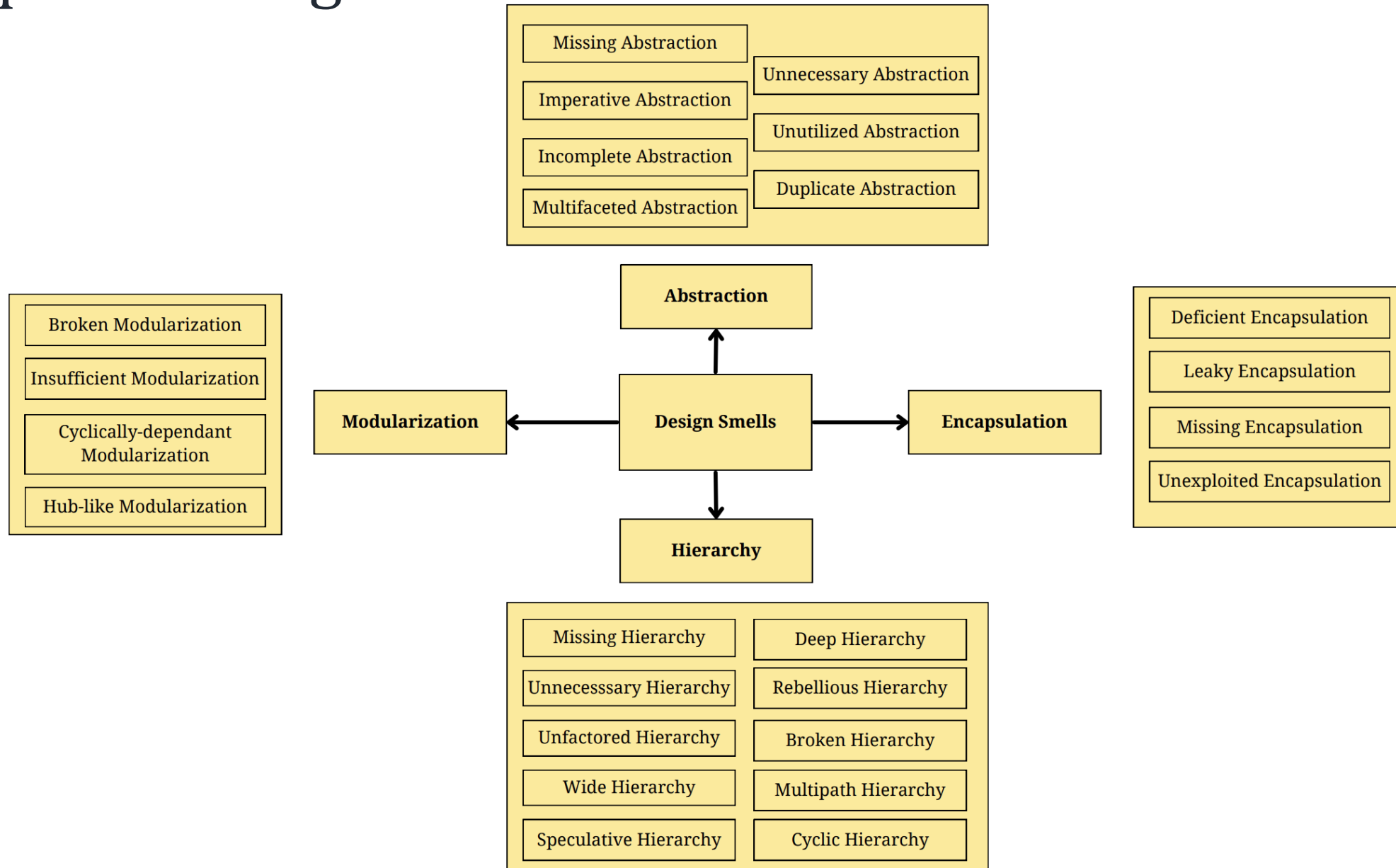INFORMATION TECHNOLOGY
HYDERABAD
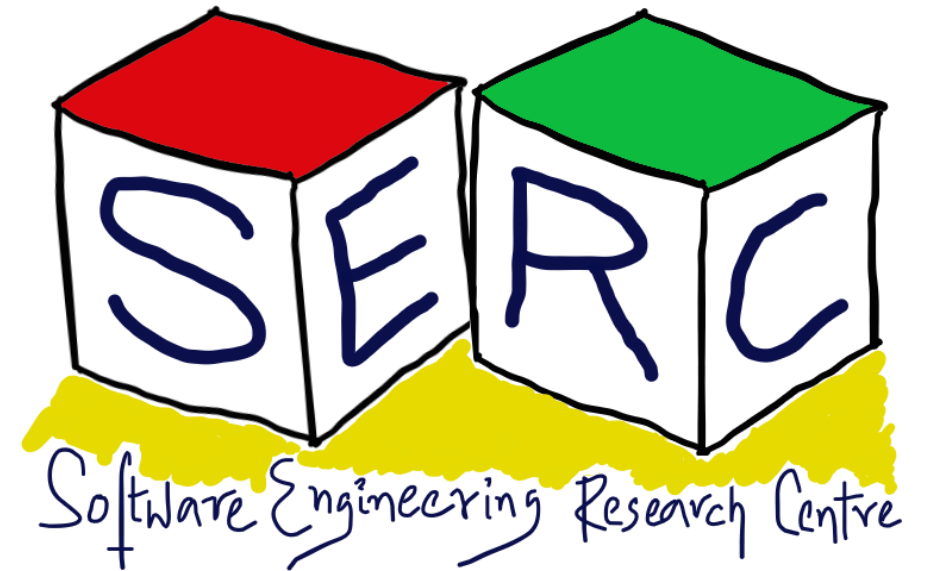
Acknowledgements

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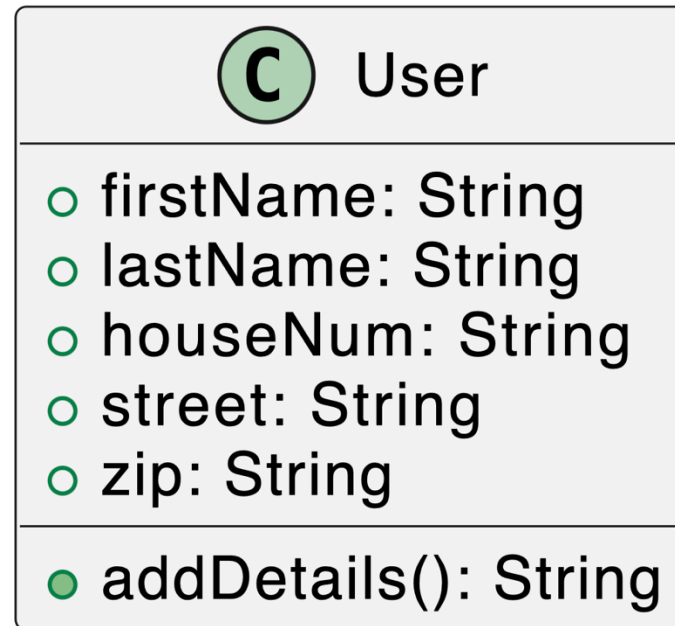-- Karthik Vaidhyanathan

Sources:

1. Refactoring, Improving the design of existing code, Martin Fowler et al., 2000
2. Refactoring for Software design Smells, Girish Suryanarayana et al.
3. martinfowler.com
4. Few articles by Ipek Ozkaya and Robert Nord, SEI, CMU

Types of Design Smells

Abstraction
- Missing Abstraction
- Unnecessary Abstraction
- Imperative Abstraction
- Unutilized Abstraction
- Incomplete Abstraction
- Duplicate Abstraction
- Multifaceted Abstraction

Modularization
- Broken Modularization
- Insufficient Modularization
- Cyclically-dependant Modularization
- Hub-like Modularization

Design Smells

Encapsulation
- Deficient Encapsulation
- Leaky Encapsulation
- Missing Encapsulation
- Unexploited Encapsulation

Hierarchy
- Missing Hierarchy
- Deep Hierarchy
- Unnecesssary Hierarchy
- Rebellious Hierarchy
- Unfactored Hierarchy
- Broken Hierarchy
- Wide Hierarchy
- Multipath Hierarchy
- Speculative Hierarchy
- Cyclic Hierarchy

3

Missing Abstraction – Example Scenario

Scenario: Consider the e-bike system which requires to store address of every user

User

- firstName: String
- lastName: String
- houseNum: String
- street: String
- zip: String

- addDetails(): String

Data clumps!!

Missing Abstraction – Example Refactoring

Solution: Refactor the design, move collection of primitive types and form a separate class
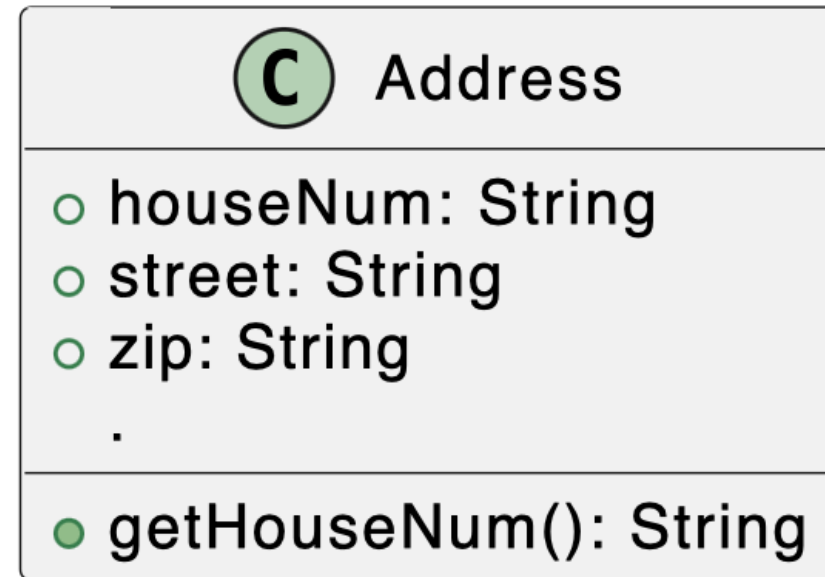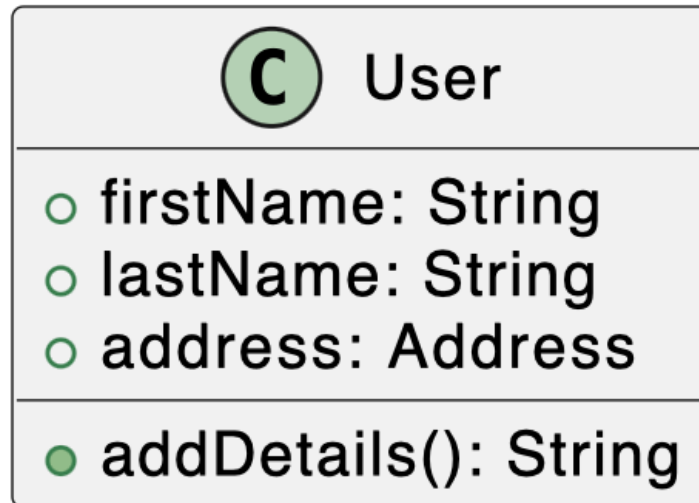
Abstraction Smell – Missing Abstraction

Indication: Usage of clumps of data or strings used instead of class or interface

Rationale: Abstraction not identified and represented as primitive types

Causes: Inadequate design analysis, lack of refactoring, focus on minor performance gains

Impact: Affects understandability, extensibility, reusability, .

Abstraction Smell – Imperative Abstraction

Scenario: Consider the e-bike system where students have to perform different operations on their wallet
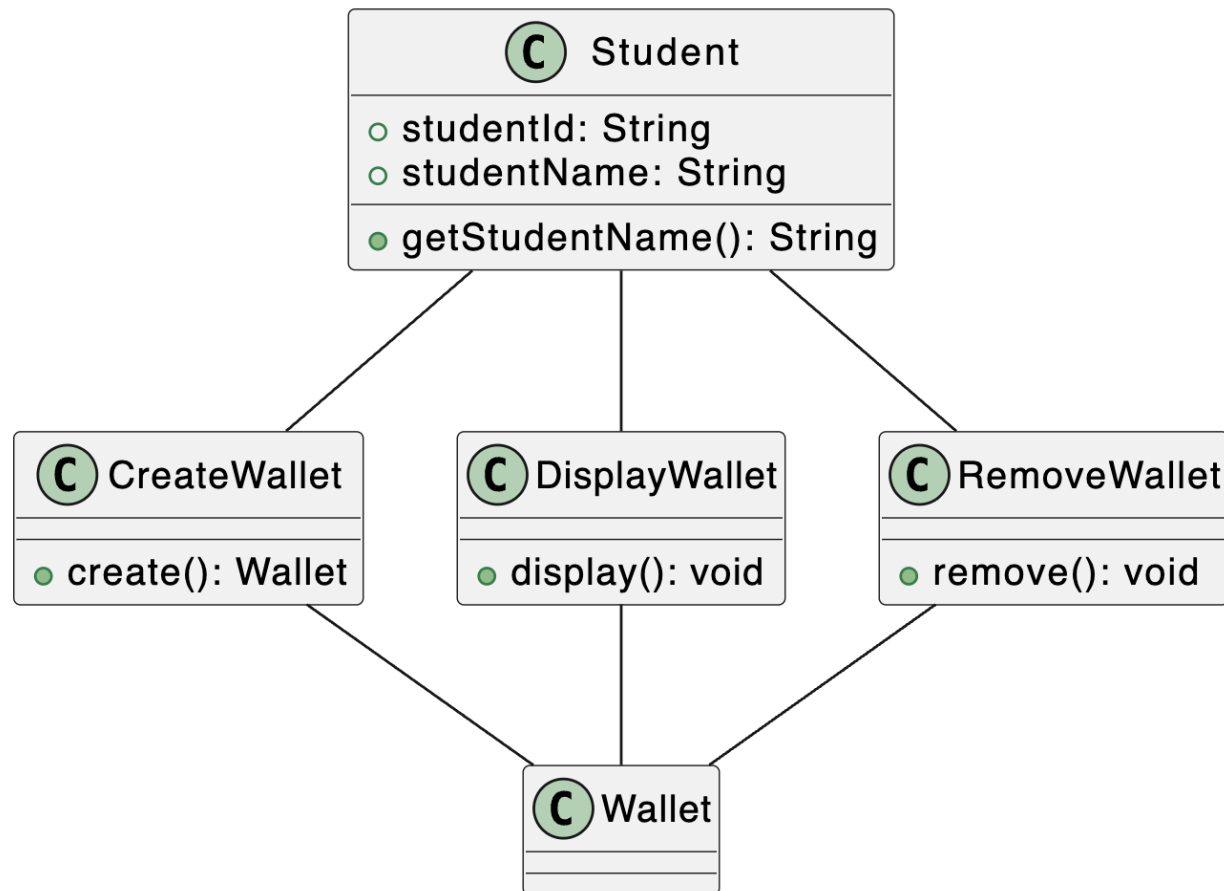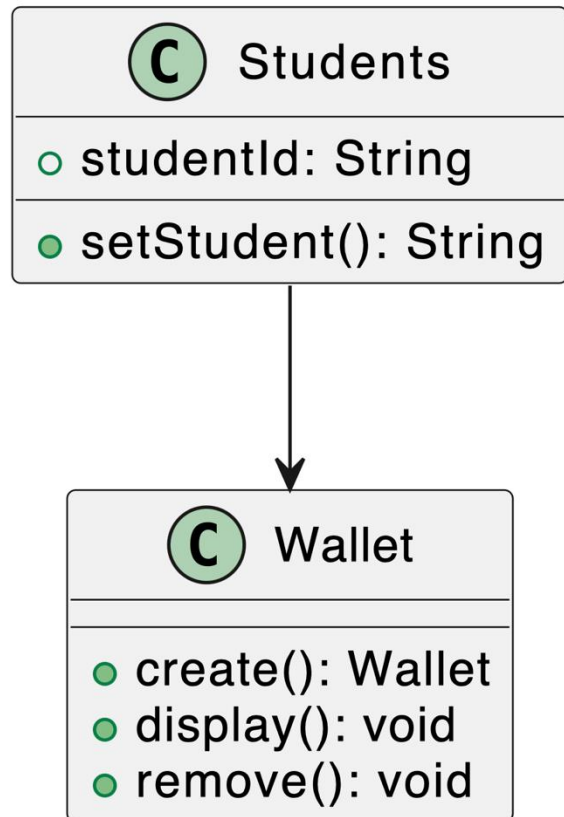

What all problems do you foresee?

Wallet will have different properties

7

Abstraction Smell – Example Refactoring

Solution: Refactor the design, move the functions into one class and bundle it with data

Remember abstraction is all about generalization
And specification of common and important characteristics!!

8

Abstraction Smell – Imperative Abstraction

Indication: Operation is turned into a class. A class that has only one method defined in it

Rationale: Defining functions explicitly as classes when data is located somewhere violates OOPS principles. Increases complexity, reduce cohesiveness

Causes: Procedural thinking (capture the bundled nature)

Impact: Affects understandability, extensibility, testability, reusability..

Abstraction - Enablers

- Crisp boundary and identity
  - Make abstractions when necessary and have clear boundaries
- Map domain entities
  - Vocabulary mapping from problem domain to solution domain
- Ensure coherence and completeness
  - Completely support a responsibility, don't spread across
- Assign Single and Meaningful Responsibility
  - Each abstraction has unique and non-trivial responsibility
- Avoid Duplication
  - The abstraction implementation and the name appears only once in design
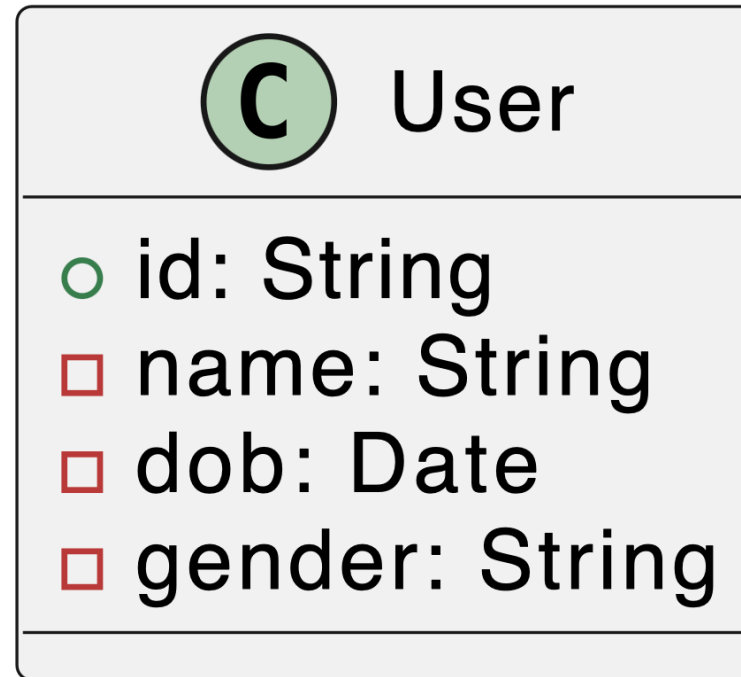
Encapsulation Smell – Deficient Encapsulation

Scenario: Consider the e-bike system where user details like DOB, gender, etc. are public
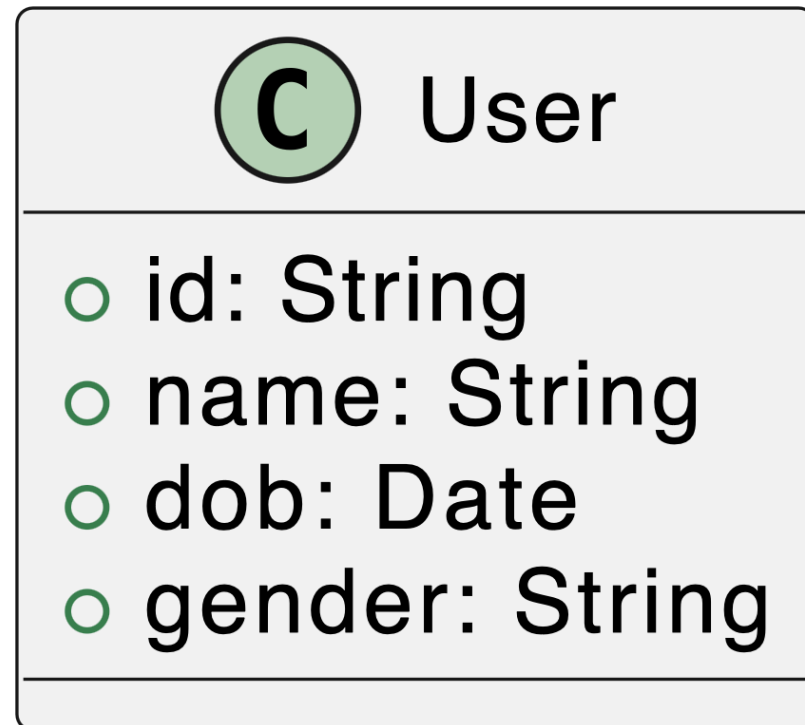
Encapsulation Smell – Example Refactoring

Solution: Refactor the design, modify the access specifiers without affecting others

Encapsulation Smell – Deficient Encapsulation

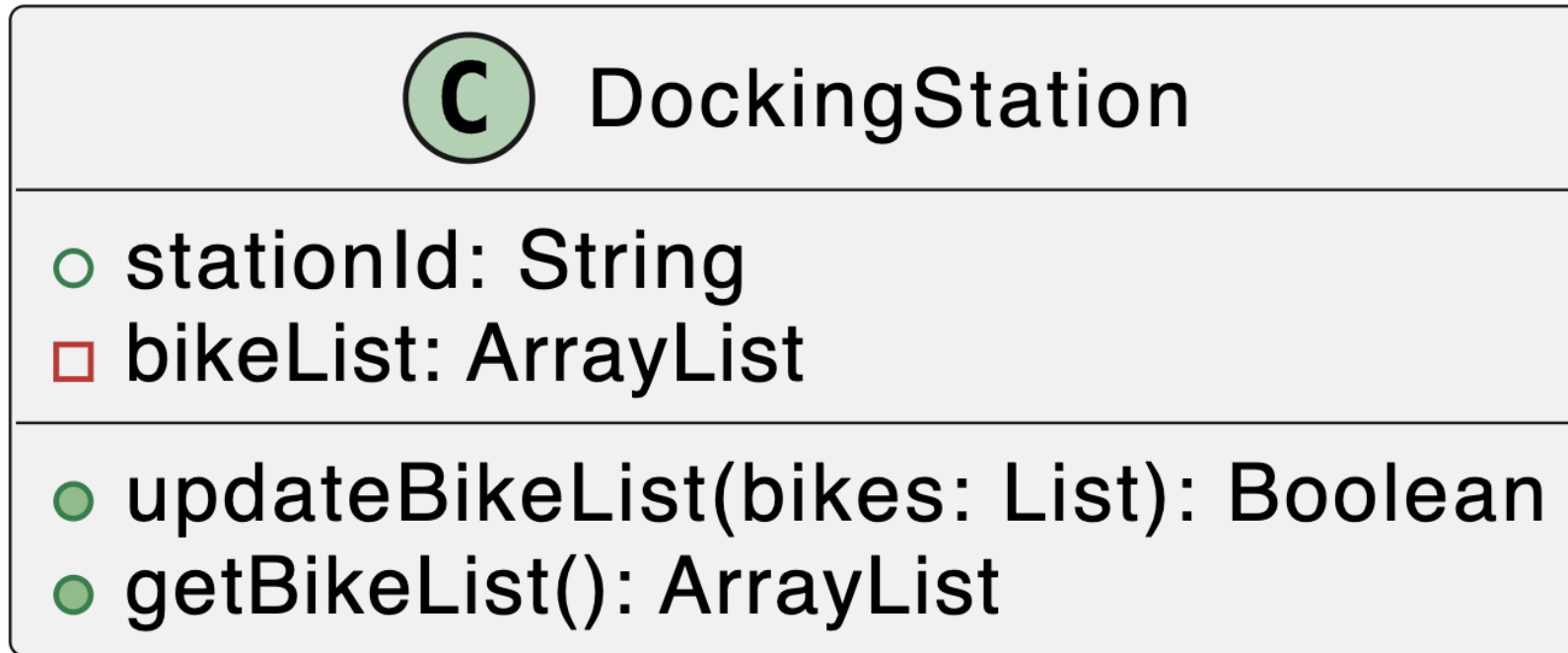Indication: One or more members is not having required protection (eg: public)

Rationale: Exposing details can lead to undesirable coupling. Each change in abstraction can cause change in dependent members

Causes: Easier testability, procedural thinking (expose data as global variables), quick fixes

Impact: Affects changeability, extensibility, reliability,...

Encapsulation Smells – Leaky Encapsulations

Scenario: Consider the e-bike system where the docking station class provides list of bikes parked in that station

DockingStation

- stationId: String
- bikeList: ArrayList

- updateBikeList(bikes: List): Boolean
- getBikeList(): ArrayList

Encapsulation Smell – Example Refactoring

Solution: Refactor the design, make return types of public more abstract to support modifiability, ensure clients do not get direct access to change internal state
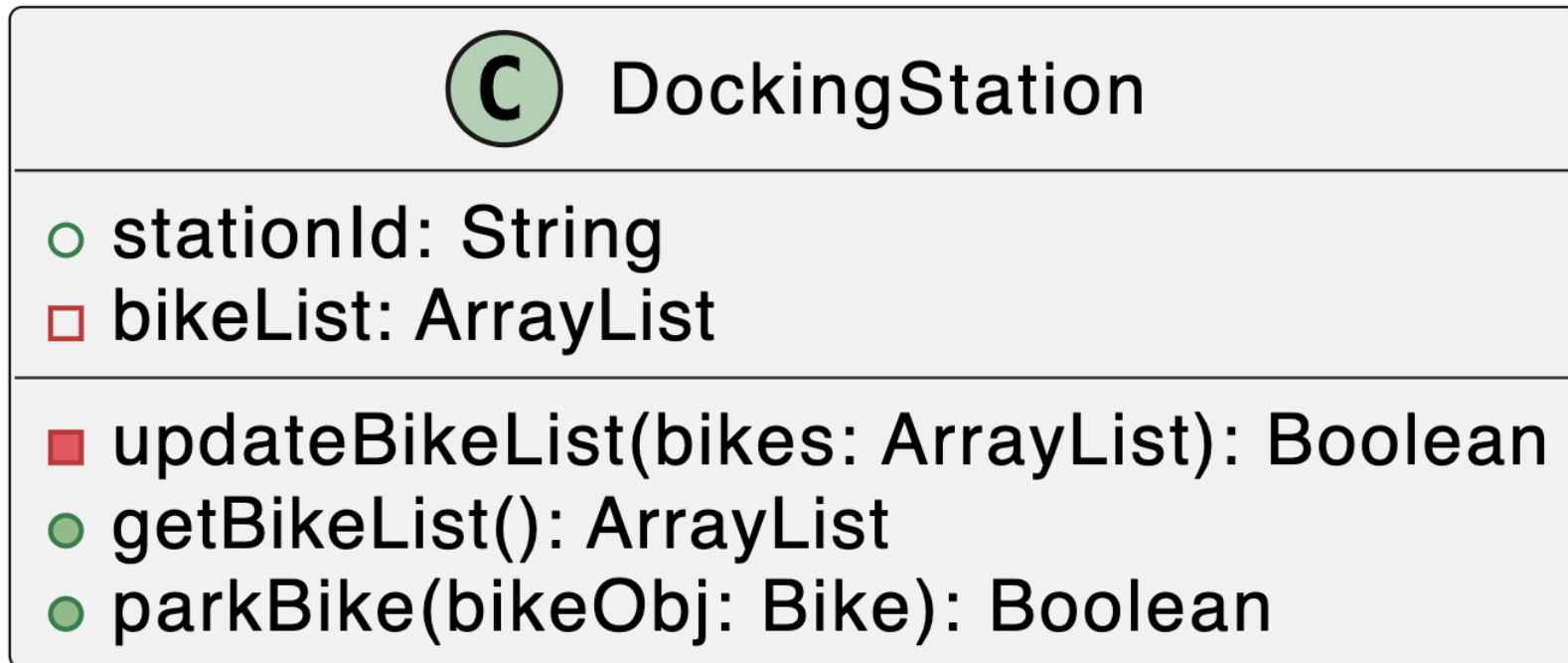
Park vehicle function can internally update the bike list

Encapsulation Smells – Leaky Encapsulations

Indication: Abstraction leaks implementation details (public methods)

Rationale: Implementation details needs to be hidden, Internal state can be corrupted due to open methods

Causes: lack of awareness, project pressure (quick hacks), too fine-grained public methods exposed (think of simple setter)

Impact: Affects changeability, reusability, Reliability

Encapsulation - Enablers

- Hide implementation details
  - Abstraction exposes only what abstraction offers and hides implementation
  - Hide data members and details on how the functionality is implemented

- Hide Variations
  - Hide implementation variations in types or hierarchies
  - Easier to make changes in abstraction implementation without affecting subclasses or collaborators
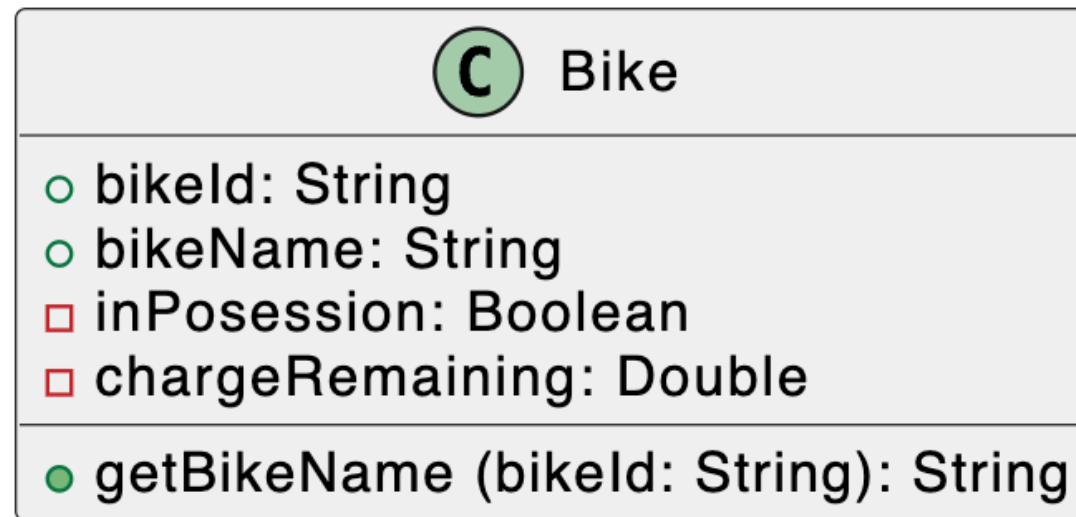
Modularization Smells – Broken Modularization

Scenario: Bike class gets all data from BikeDetails class but all operations resides in Bike Class
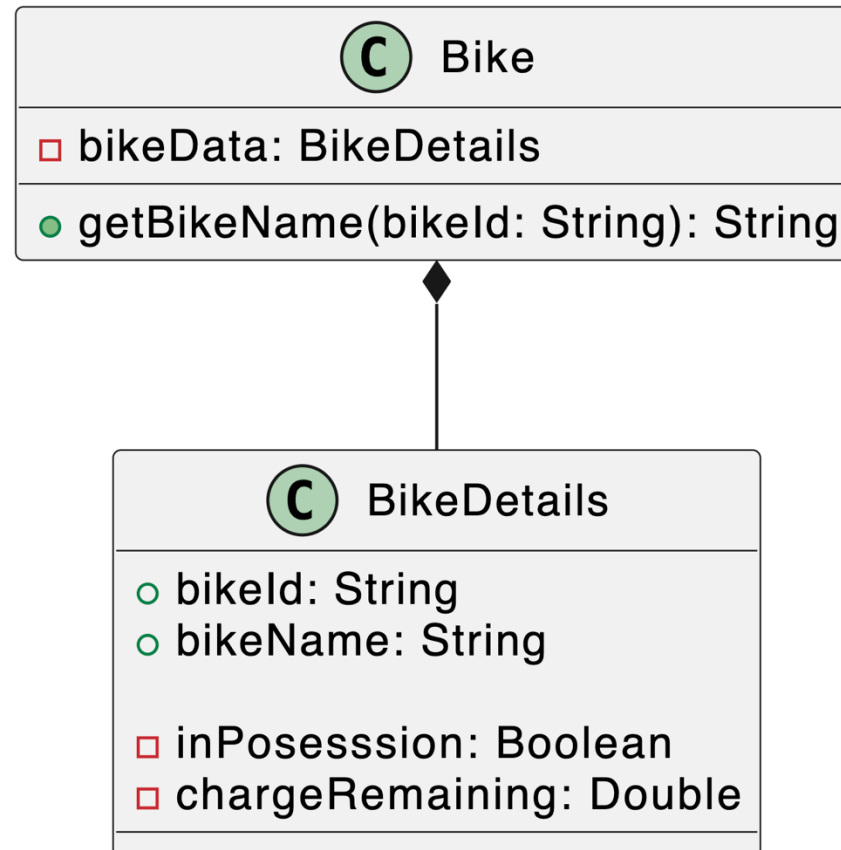
Modularization Smells – Example Refactoring

Solution: Refactor the design in such a way that the data and methods stay together as a unit. Enhancing cohesiveness is the key

Modularization Smells – Broken Modularization

Indication: Data and methods are spread across instead of being bundled

Rationale: Having data in one and methods in another results in tight coupling, violates modularity

Causes: Procedural thinking, lack of understanding of existing design

Impact: Affects changeability and extensibility, reusability, Reliability

Modularization Smells – Enablers

- Localize related data and methods
  - All the data and method related to one class should be kept in the same class
- Abstractions should of manageable size
  - Ensure classes are of manageable size – mainly affects maintainability, extensibility and understandability
- Ensure there are no cyclic dependencies
  - Graph of relationships between classes should be acyclic
- Limit Dependencies
  - Create classes with low fan-in and low fan out
    - Fan-in: number of incoming dependencies
    - Fan-out: number of outgoing dependencies
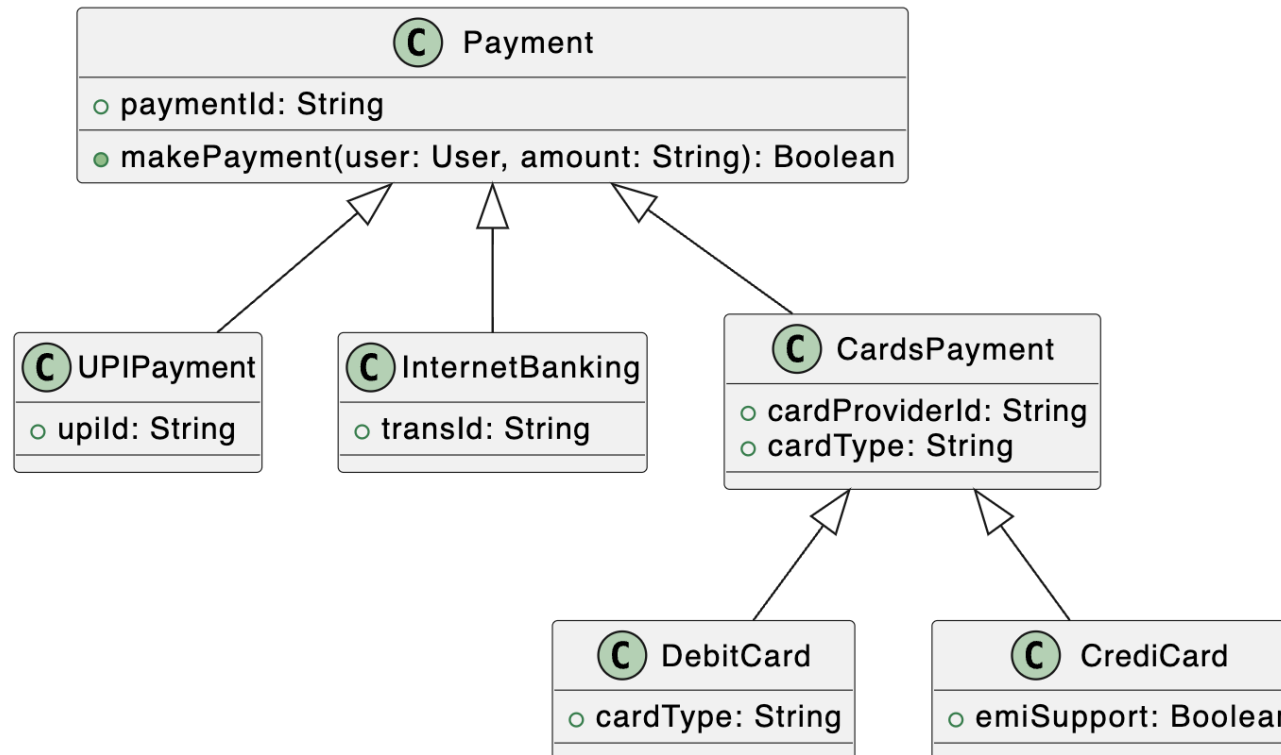
Hierarchy Smells – Missing Hierarchy

Scenario: In the e-vehicle scenario, user can pay in any mode of payment
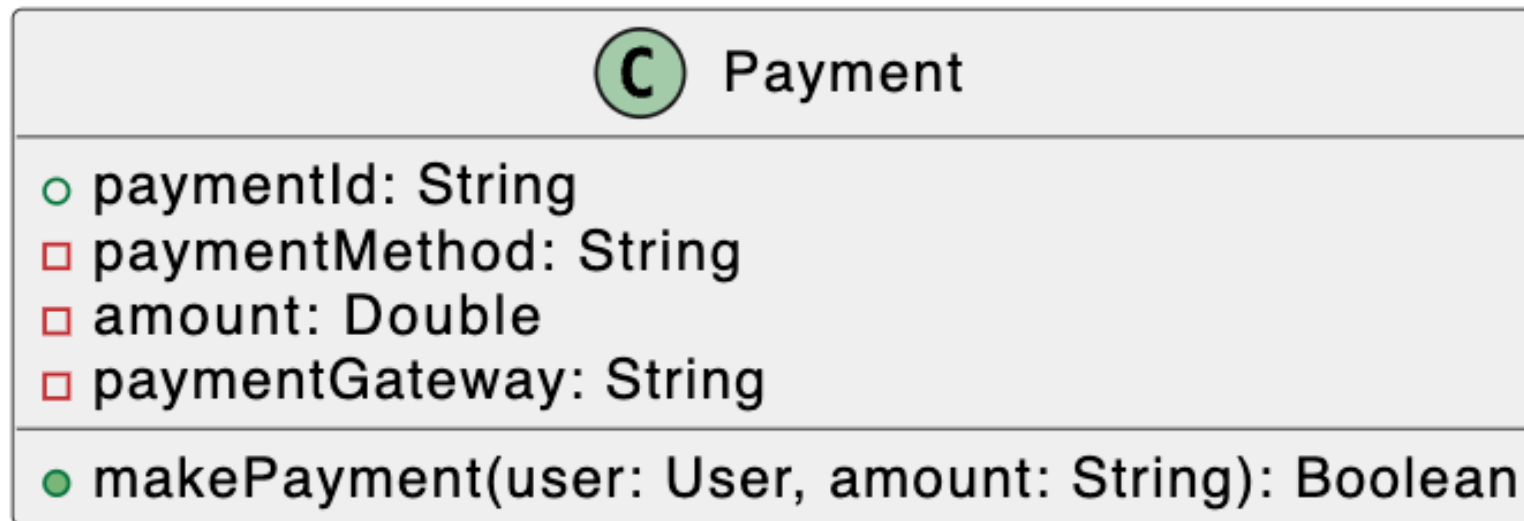
One way to support different types of payment is to write them inside makePayment function

Hierarchy Smells – Example Refactoring

Solution: Refactor by creating hierarchies based on the behavior changes that comes under payment function. Put the common parts in parent class (think about abstract class or interfaces as well)

Note: DebitCard and CreditCard needs to be Specialized and generalized into Cards only if They have enough variation points

Hierarchy smells – Missing Hierarchy

Indication: Using if conditions to manage behavior variations instead of creating hierarchy

Rationale: Using chained if-else or Switch indicates issues with handling variations. Commonality among the types can also be used

Causes: "simplistic design", procedural approach, overlooking inheritance

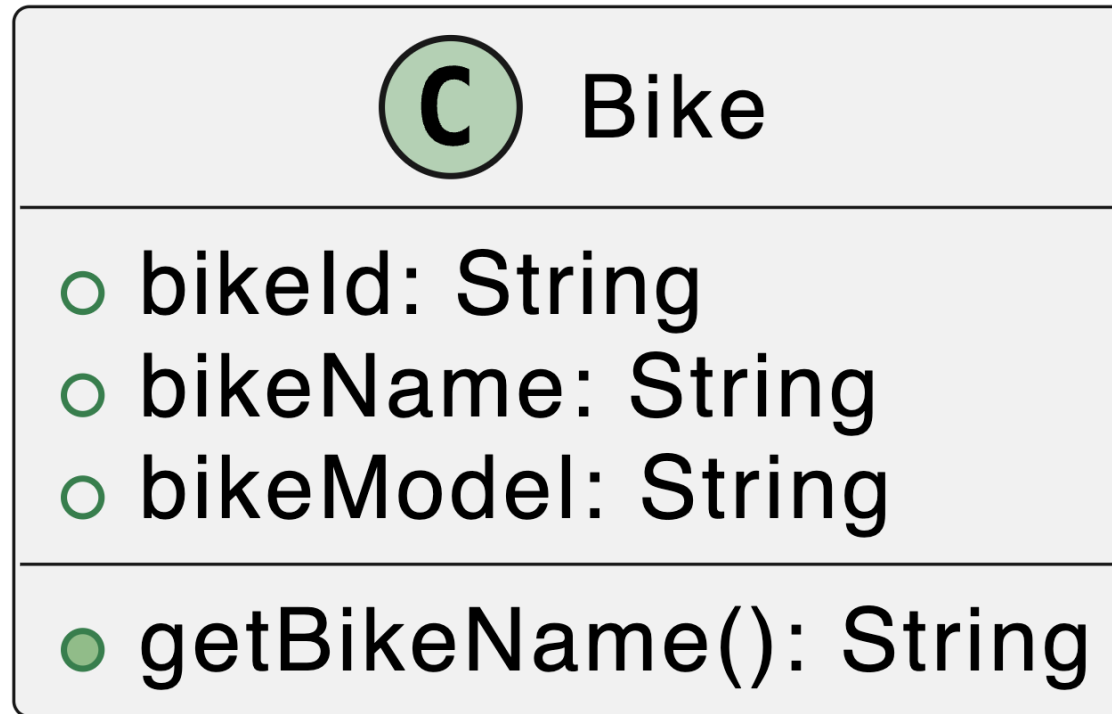Impact: Reliability, Testability, understandability, extensibility,..

Hierarchy smells – Example Scenario

Scenario: Each bike can be of different model resulting in different design (shape, colour, etc.)
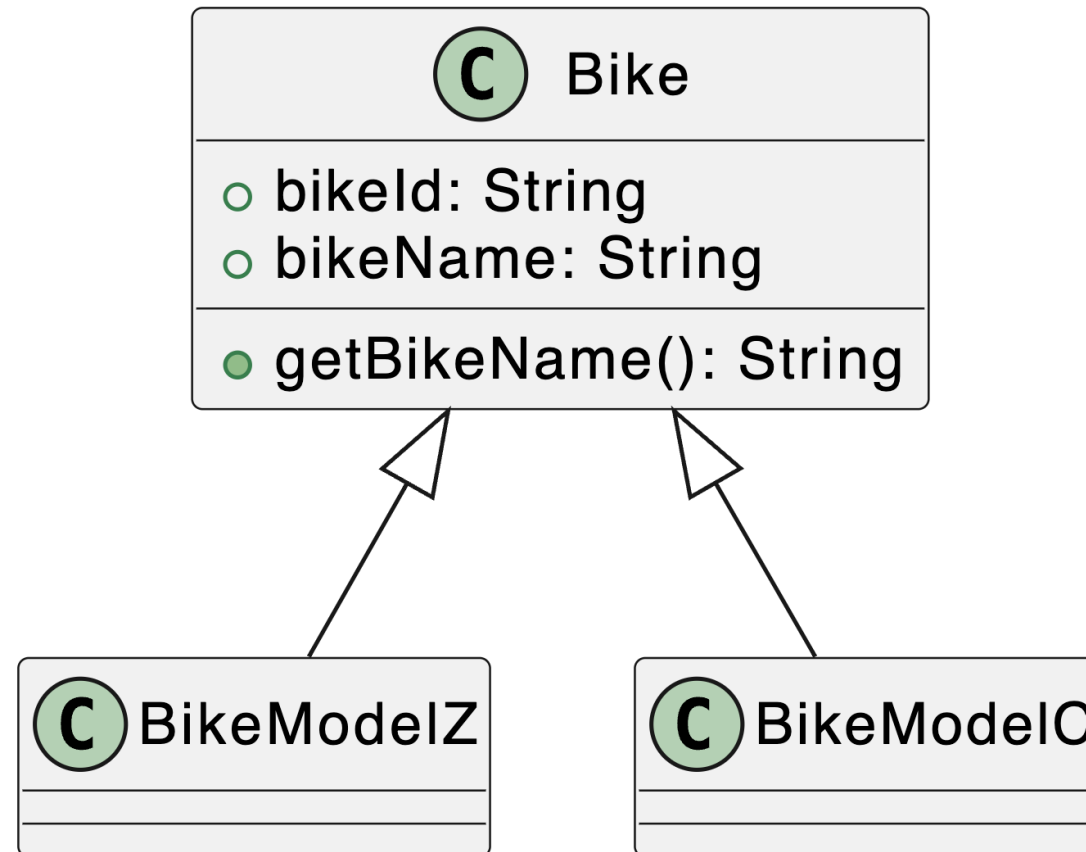
Hierarchy smells – Refactoring

Solution: Remove hierarchy and transform subtypes into instance variables

Hierarchy smells – Unnecessary Hierarchy

Indication: Inheritance has been applied needlessly for a particular context

Rationale: The focus should be more on capturing commonalities and variation in behavior than data. Violation results in unnecessary hierarchy

Causes: subclassing instead of instantiating, taxonomy mania (overuse of inheritance)

Impact: Understandability, Extensibility, Testability..

Hierarchy Smells - Enablers

- Apply meaningful classification
  - Identify commonalities and variations – Classify into levels
- Apply meaningful generalization
  - Identify common behavior and elements to form supertypes
- Ensure Substitutability
  - Reference of supertype can be substituted with objects of subtypes
- Avoid redundant paths
  - Avoid redundant paths in inheritance hierarchy
- Ensure proper ordering
  - Express relationships in a consistent and orderly manner

Some General Observations

- Analyze your design
  - Is this abstraction enough?
  - Is there some responsibility overload?
  - Have we made use of the right set of access modifiers?
  - Only expose what is necessary
  - Ensure high cohesiveness and loose coupling
  - Create hierarchies whenever necessary (only when necessary)
- Always remember, refactoring is not a one-time process
- The more it is delayed, the more debt is incurred!
- Combination of design smells exists
- Code can serve as good indicators of design smells – Code also smells!

Next up: Code Smells and Code Metrics!!

Thank You

Course website: karthikv1392.github.io/cs6401_se

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