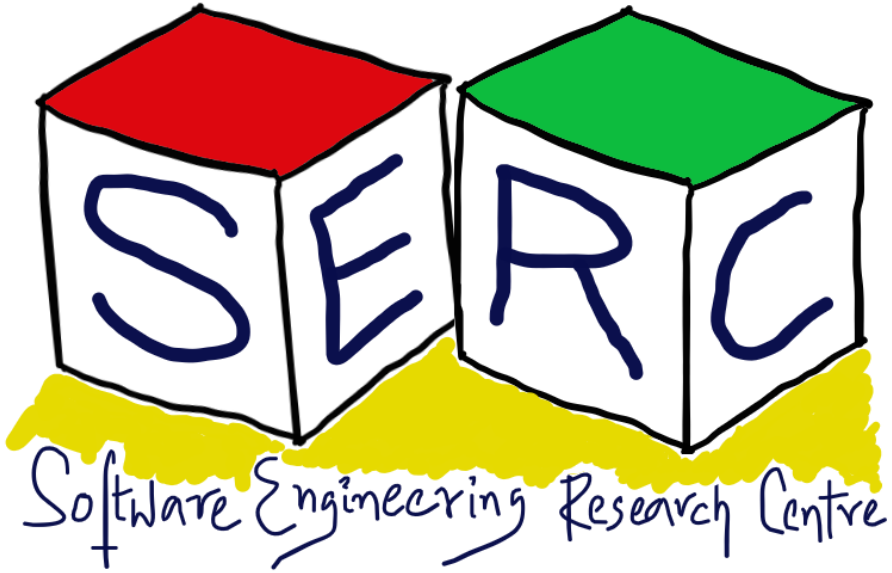
Refactoring: An Introduction

CS6.401 Software Engineering



Dr. Karthik Vaidhyanathan

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

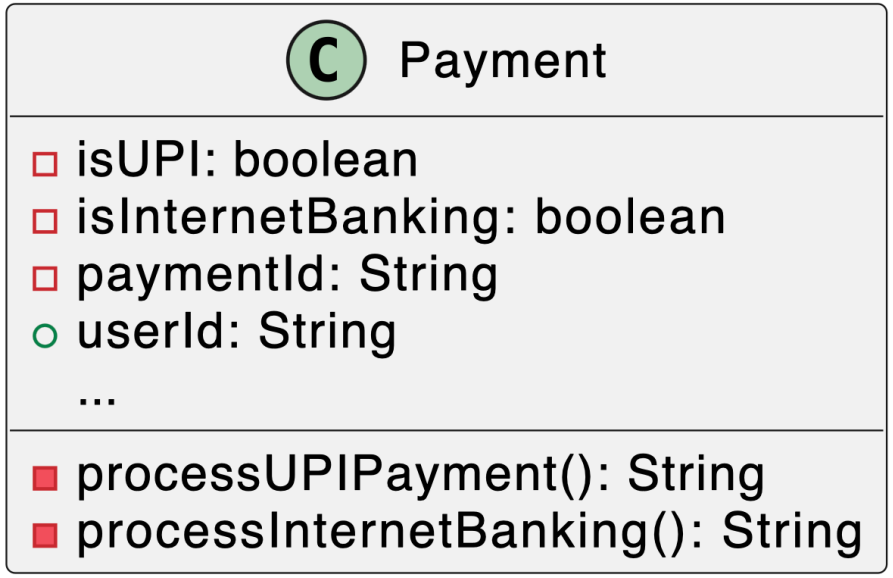
Sources:

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

As a program is evolved, its complexity increases unless work is done to maintain or reduce it

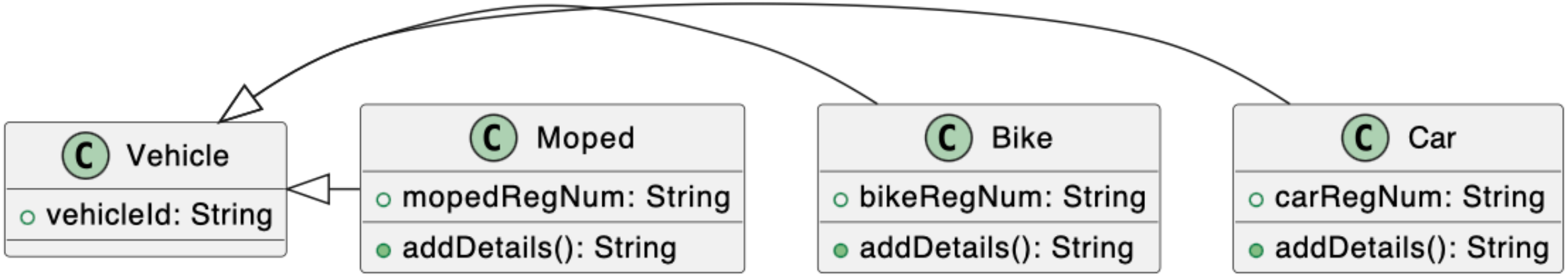
-- Lehmans' Law of Increasing Complexity

Few Examples to Begin with..



Do you see some issues here?

Few Examples to Begin with..



What about this?

Ever heard about Technical Debt?

What is Debt?

debt

/dɛt/

noun

a sum of money that is owed or due.
"I paid off my debts"

Similar: bill account tally

Source: Google dictionary - Oxford, entrepreneur.com

Technical Debt

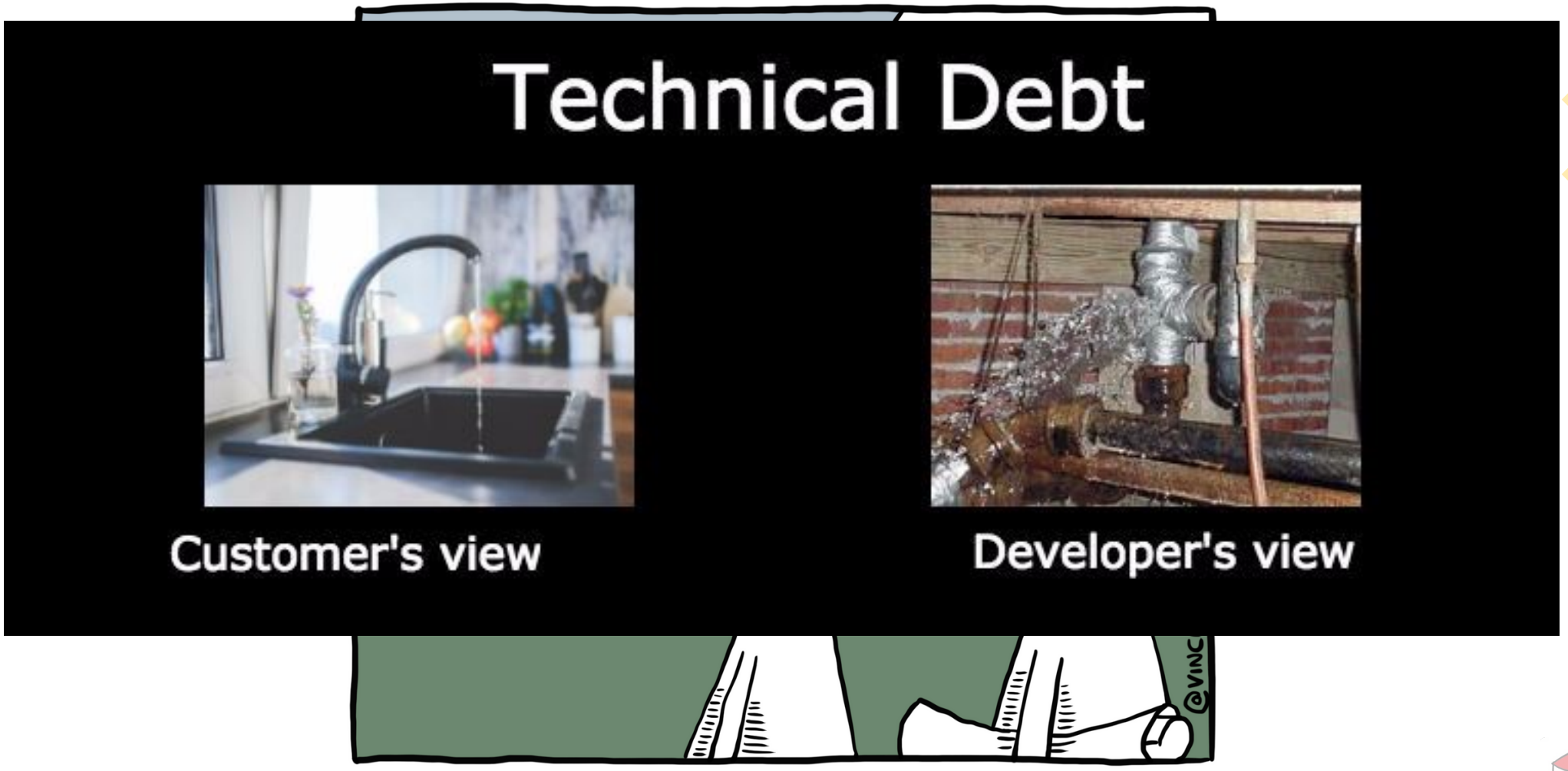
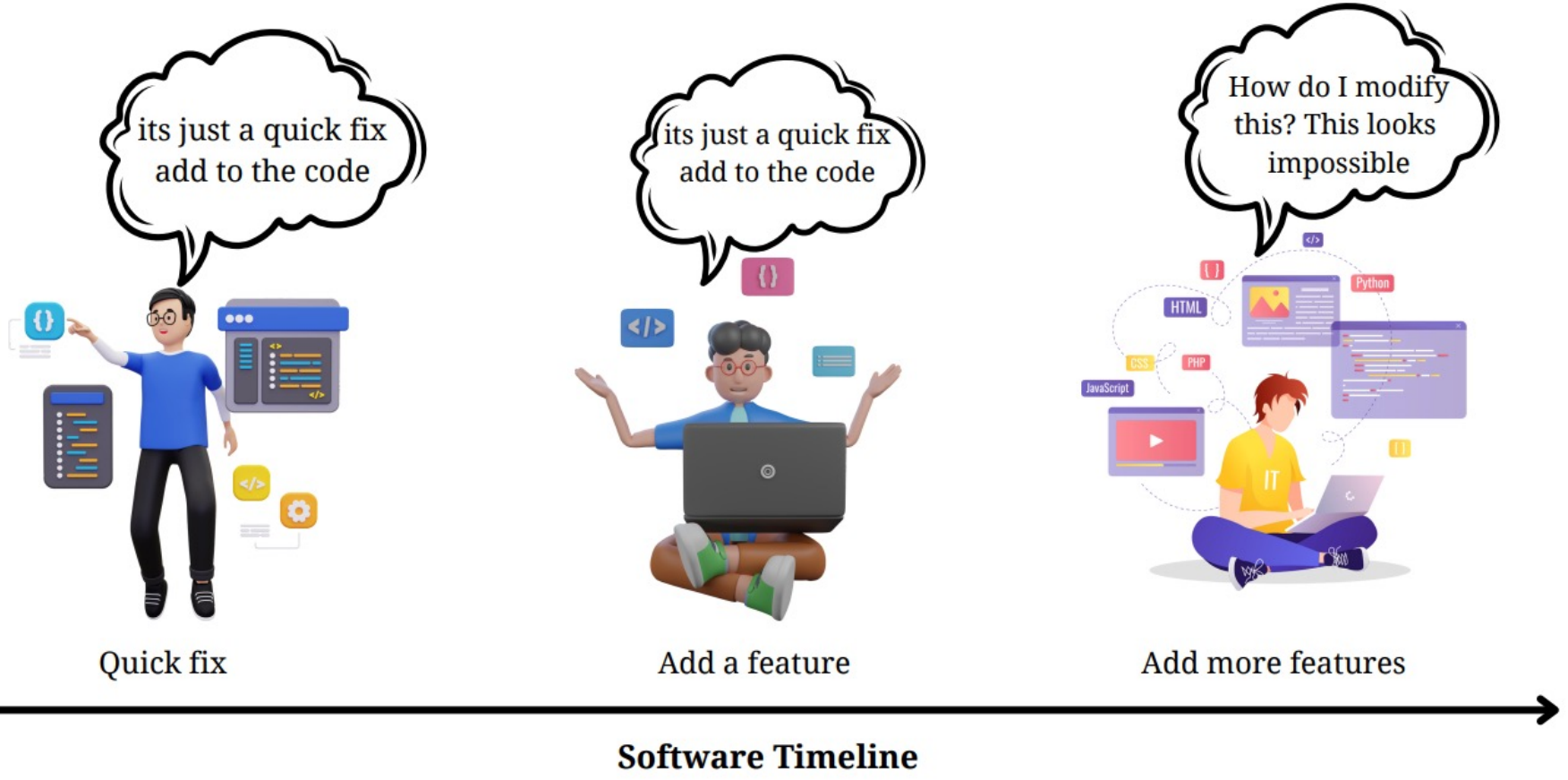


Image source: medium, google images

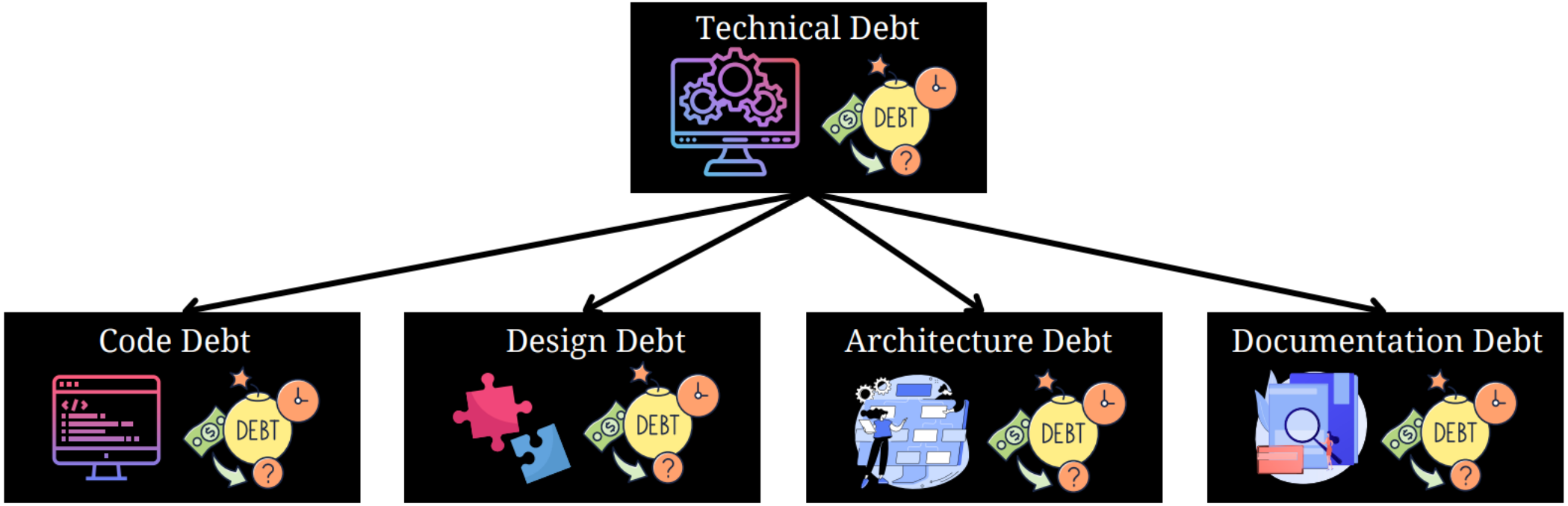
Technical Debt - Definition

Technical debt is the **debt that accrues** when you knowingly or unknowingly make **wrong or non-optimal design decisions**

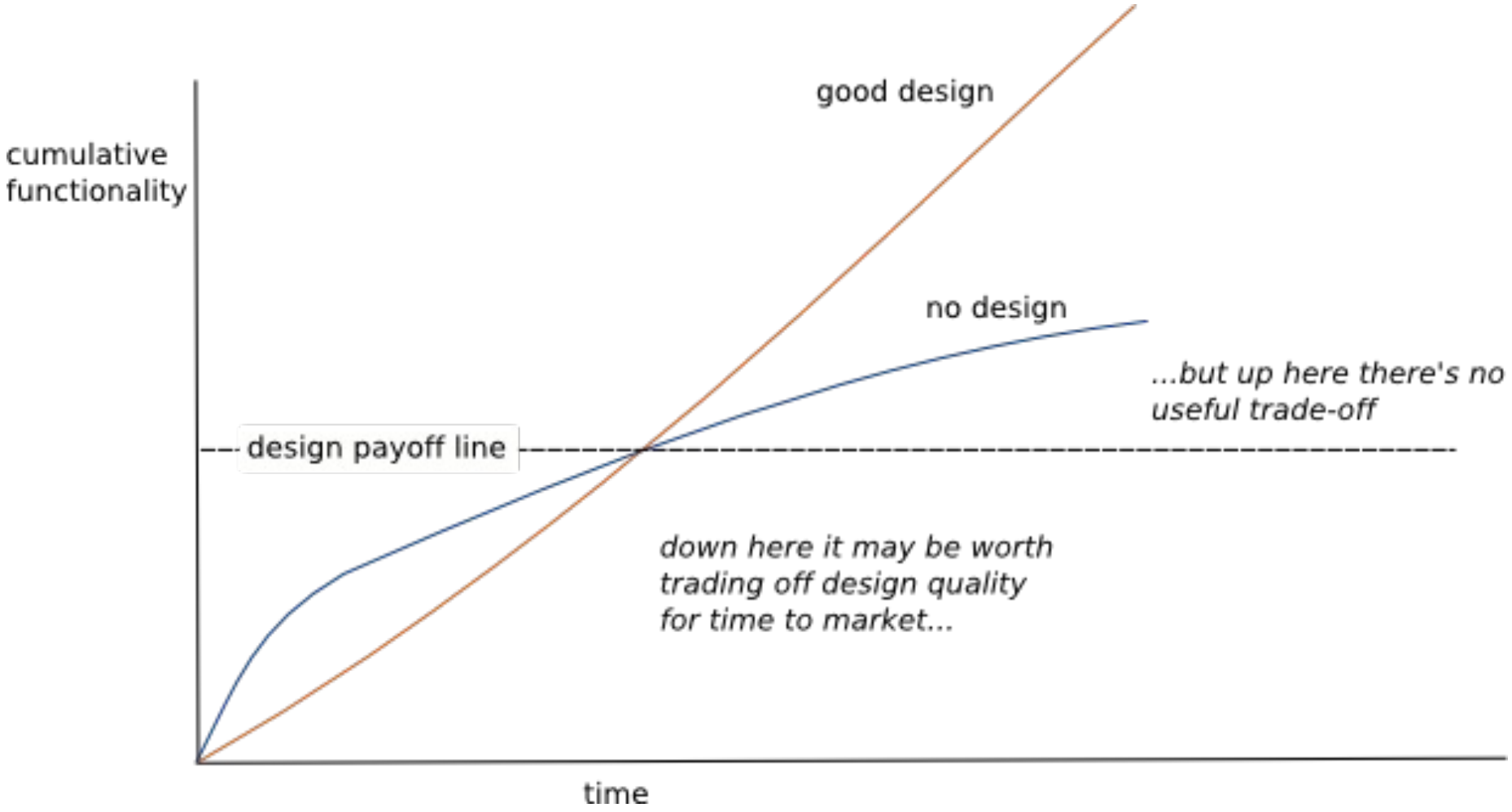
Metaphor coined by *Ward Cunningham*, 1992



Types of Technical Debt



Design Stamina Hypothesis



source: martinfowler.com

Impact of Technical Debt

"One large North American bank learned that its more than 1,000 systems and applications together generated over $2 billion in tech-debt costs" - McKinsey

- Interest is very much compounding in nature – changes has to be done on already existing debt
- Cost of Change becomes extremely high!
- Affects morale of development team
- Huge impact on progress of the business – product and feature delays
- Often considered as the digital dark matter!

Source: https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/demystifying-digital-dark-matter-a-new-standard-to-tame-technical-debt

Impact of Technical Debt – An Example Scenario

A successful company in the maritime equipment industry successfully evolved its products for 16 years, in the process amassing 3 million lines of code. Over these 16 years, the company launched many different products, all under warranty or maintenance contracts; new technologies evolved; staff turned over; and new competitors entered the industry.

The company's products were hard to evolve. Small changes or additions led to large amounts of work in regression testing with the existing products, and much of the testing had to be done manually, over several days per release. Small changes often broke the code, for reasons unsuspected by the new members of the development team, because many of the design and program choices were not documented.

What were some things they could have done right?

Source: Ozkaya, I. and Nord, R. . 2019: Managing the Consequences of Technical Debt: 5 Stories from the Field. Carnegie Mellon University's Software Engineering Institute Blog,. https://doi.org/None (Accessed January 13, 2023)

Impact of Technical Debt – Another Case

Southwest Airlines: 'Shameful' Technical Debt Bites Back

BY: RICHI JENNINGS ON JANUARY 5, 2023 — 0 COMMENTS

Welcome to *The Long View*—where we peruse the news of the week and strip it to the essentials. Let's work out **what really matters**.

20 Years of Neglect Led to 'Meltdown'

Last month's débâcle of canceled flights was caused by decades of technical debt. That's the analysis of Columbia University professor **Zeynep Tufekci**.

Analysis: SWA needs a cloud burst

Although there were several contributing factors, a lack of scalability in a critical crew scheduling system led to days of near-total paralysis: In many cases, the staff were in the right place to fly and crew the planes, but the *SkySolver* system had no way of knowing that. Making things worse, manual fallbacks collapsed under the **weight of the workload**.

The answer:… employee scheduling software that debuted around the same time as the Xbox 360 and PlayStation 3. … Southwest pilots have reportedly begged company executives to update the "antiquated" systems since at least 2015.

Eventually someone has to pay for the debt!!

Source: https://devops.com/southwest-technical-debt-richixbw/

Reasons for Technical Debt

Everyone in the decision making could be blamed – Architects, developers, managers.. but that doesn't end there. There are many other reasons..

- Schedule pressure – Copy paste programming
  - Its not always about getting the syntax right and making something work
- Lack of skilled designers – Poor applications of design principles
  - Lack of awareness about best practices
  - Leading in the wrong direction
- Lack of awareness of key indicators and refactoring - Design issues
  - Periodic review of design and making changes can go a long way!!

Managing Technical Debt

- Increase awareness about tech debt
  - Being aware is the best start
  - Create goals keeping this in mind
- Detect and repay tech debt systematically
  - Identify instances of debt (huge impact)
  - Create systematic plan on recovery
- Prevent accumulation of tech debt
  - Once under control, prevent further accumulation
  - Perform regular monitoring
- Companies should allocate some budget for tech debt

Image source: xkcd

Key Major Questions

1. Why do even good developers write bad software?
2. How do we fix our software?
3. How to know if the software is “bad” even when its working fine?

Refactoring!

“Any fool can write code that a computer can understand. Good Programmers write code that humans can understand”

Martin Fowler
Thoughtworks

Image source: thoughtworks

What is Refactoring?

It is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour

-- Martin Fowler

Image source: imageflip.om

What is Refactoring?

- Refactoring is not always a clean up of code!
- Goal is to make software easier to understand and modify
- Think of performance optimization
- Refactoring does not or should not change behavior – No change to external user [Changing hats]
- Not always same as:
  - Adding features
  - Debugging code
  - Rewriting code

Image source: imageflip.om

When to Refactor?

- Follow the rule of three
  - First time, just get it done
  - Second time to do something similar, duplicate
  - Third time, just refactor
- Refactor when you add a function (feature)
  - When adding new feature, make it more effective and efficient
- Refactor when you fix a bug
  - Bug by themselves can be good indicators – Are they becoming more common?
- Refactor when you do code reviews
  - Create review groups for code reviews, new perspective may lead to refactoring

Image source: imageflip.om

Some Common Refactoring – Low Level refactoring

- IDEs provide a lot of support
- Variable/method/class renaming
- Extraction of duplicate code snippets
- Change in method signature
- Method or constant extraction
- Warnings about unused variables, parameter uses/declarations
- Auto-completion support and minimal documentation support

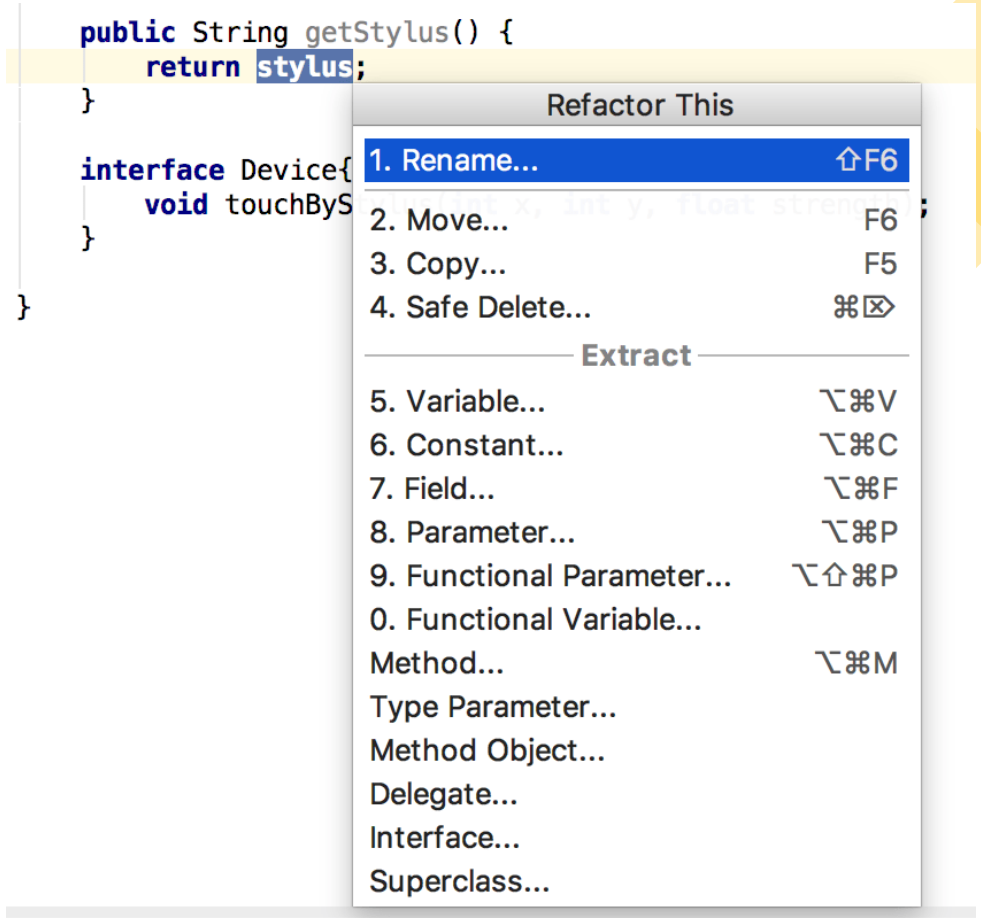


Image source: Intellij.com

High-level refactoring - Challenges

- Much more complex – has dependency on use case, context
- Risk of introducing bugs – Changes in design can introduce new issues
- Testing can become difficult – New test cases needs to be added, overall

Behavior may change [ideally not!]

- Communication of changes – Changes can be more abstract and harder to explain
- Measuring the impact – Changes can be harder to quantify

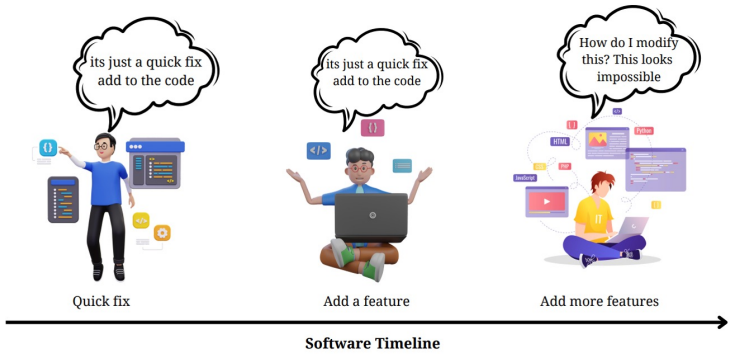
Image source: imageflip.om

Summary So Far

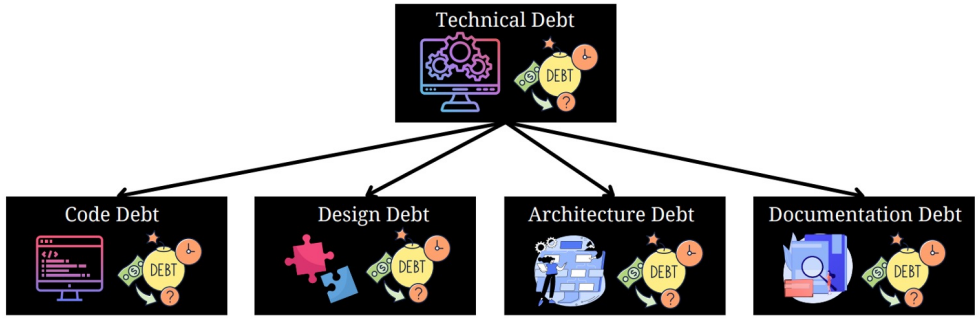
Technical Debt - Definition

Technical debt is the **debt that accrues** when you knowingly or unknowingly make **wrong or non-optimal design decisions**

Metaphor coined by *Ward Cunningham*, 1992



Types of Technical Debt



What is Refactoring?

It is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour

-- Martin Fowler

Image source: imageflip.com

High-level refactoring - Challenges

- Much more complex – has dependency on use case, context
- Risk of introducing bugs – Changes in design can introduce new issues
- Testing can become difficult – New test cases needs to be added, overall

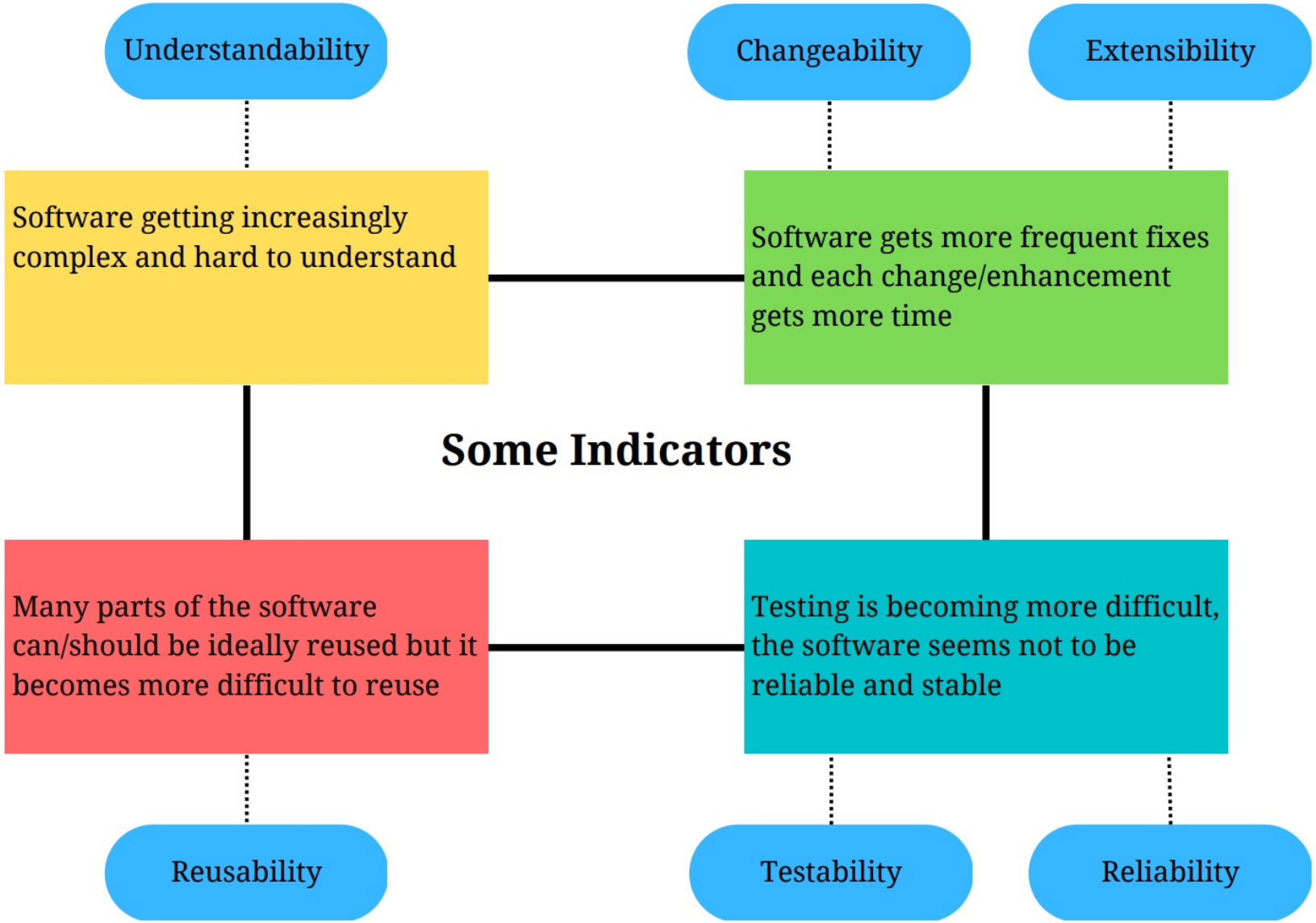
Behavior may change [ideally not!]

- Communication of changes – Changes can be more abstract and harder to explain
- Measuring the impact – Changes can be harder to quantify

Image source: imageflip.com

How to Identify Technical Debts and Refactor?

Software Quality as an Indicator



How to Refactor?

- Identify the refactoring points
- Create a refactoring plan
- Make a backup of the existing codebase: Versioning system
- Use semi-automated approach: Some tool support is always available
- Perform the refactoring
- Test if everything works like before! – Test extensively (new bugs, broken functionalities, etc.)
- Repeat the process

Remember: Refactoring is not just a one time activity!!

Refactoring Points - Things starts to rot and **Smell**

Code Smells and so does design – You heard that right!!!

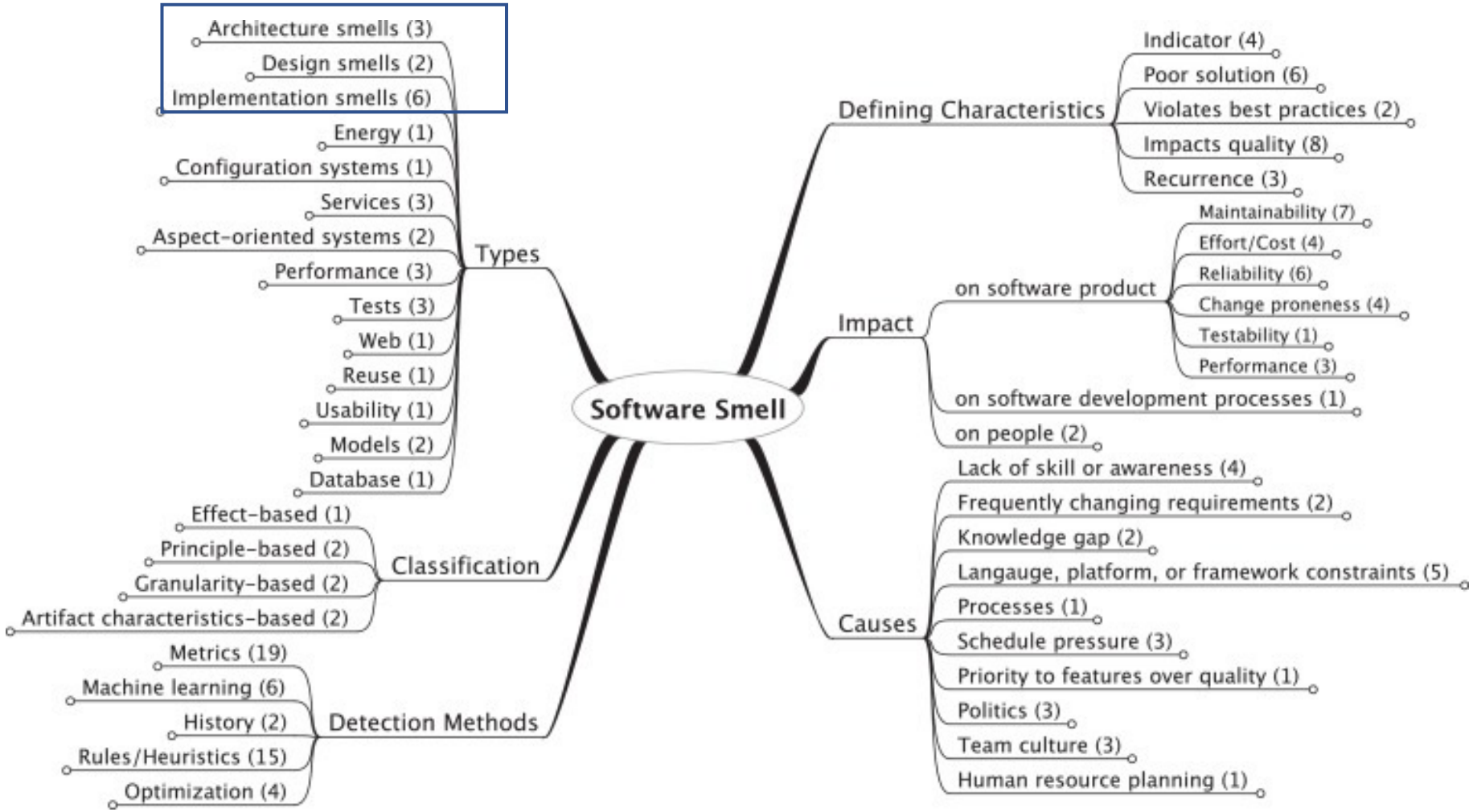
”smell”, Coined by Kent Beck in 1999

Smells are certain structures in the code that **suggest** (sometimes they scream for) the **possibility of refactoring**

A ”bad smell" describes a situation where there are hints that suggest there can be a **design problem**

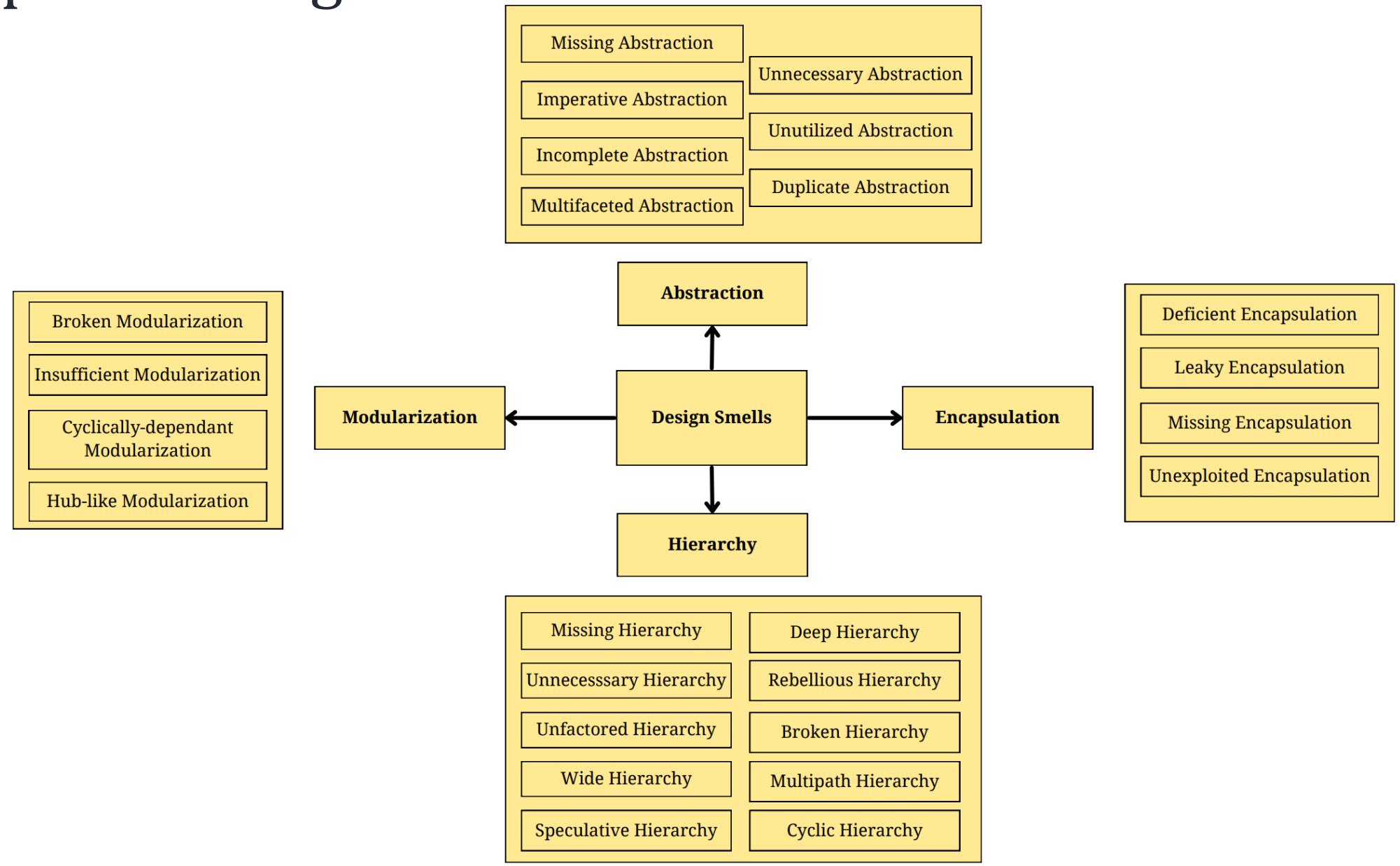
Many different definitions - https://zenodo.org/record/1066135#.Y8PcXS8RpQI

Many methods, reasons, ways to detect..



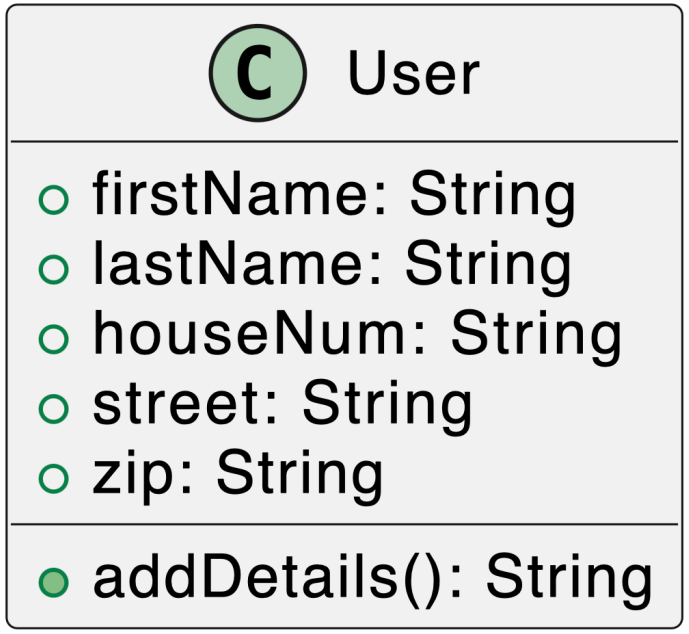
Tushar Sharma, Diomidis Spinellis, **A survey on software smells**, Journal of Systems and Software, Volume 138, 2018

Types of Design Smells



Missing Abstraction – Example Scenario

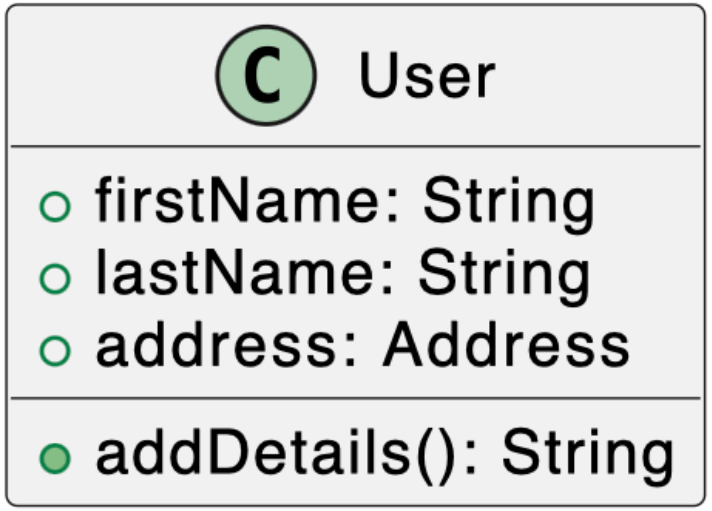
Scenario: Consider the e-bike system which requires to store address of every user

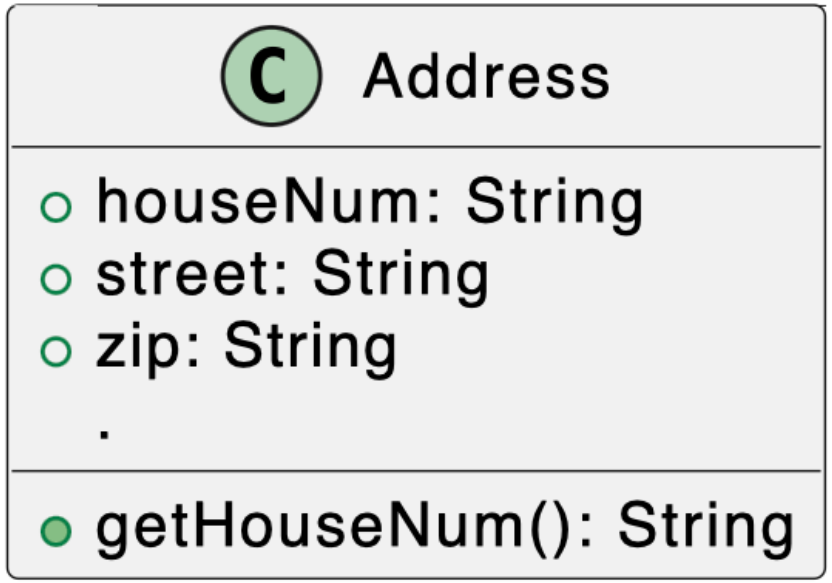


Data clumps!!

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

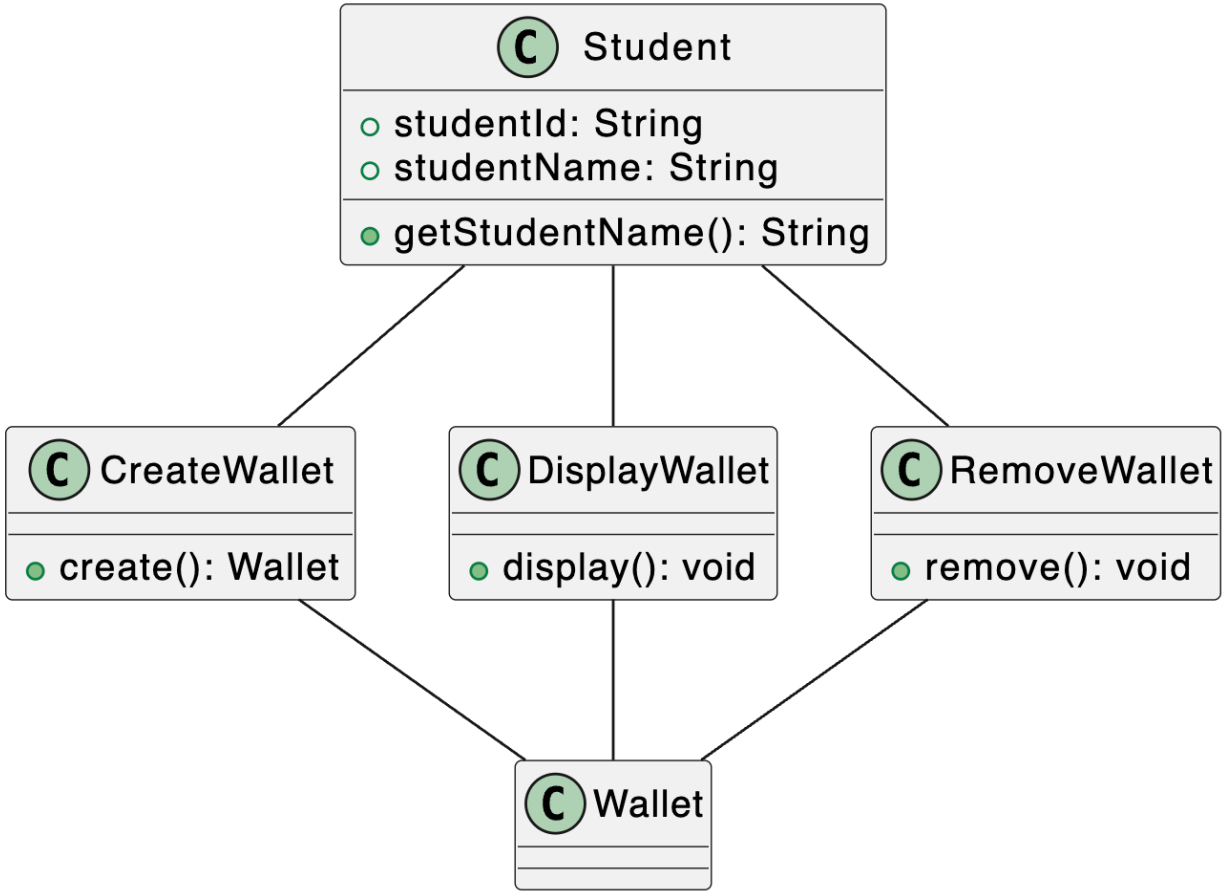
Rationale: Abstraction not identified and represented as primitive types

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

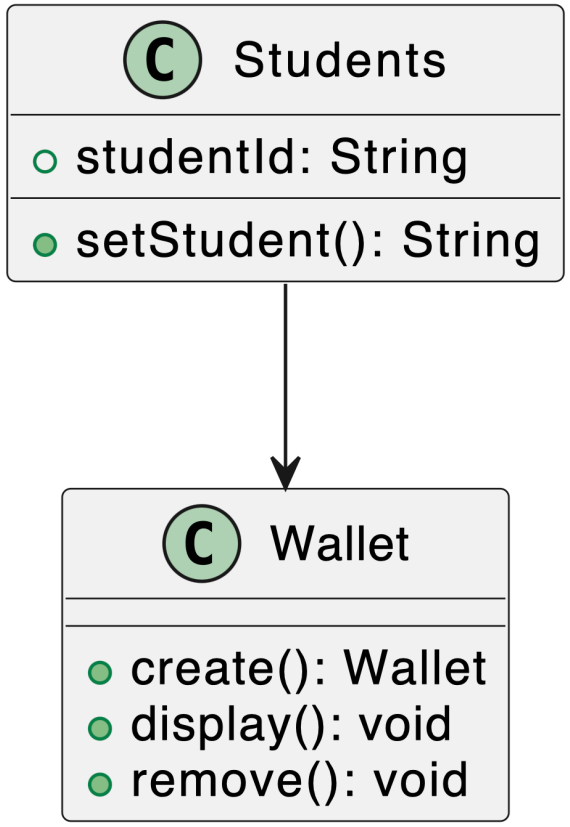


What all problems do you foresee?

Wallet will have different properties

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!!

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)

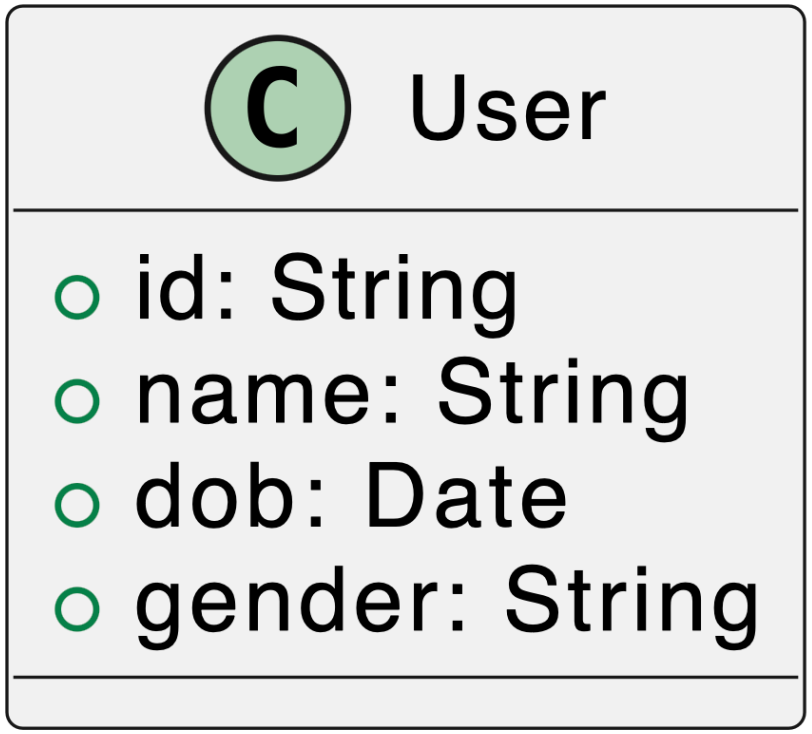
Impact: Affects understandability, extensibility, testability, reusability..

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

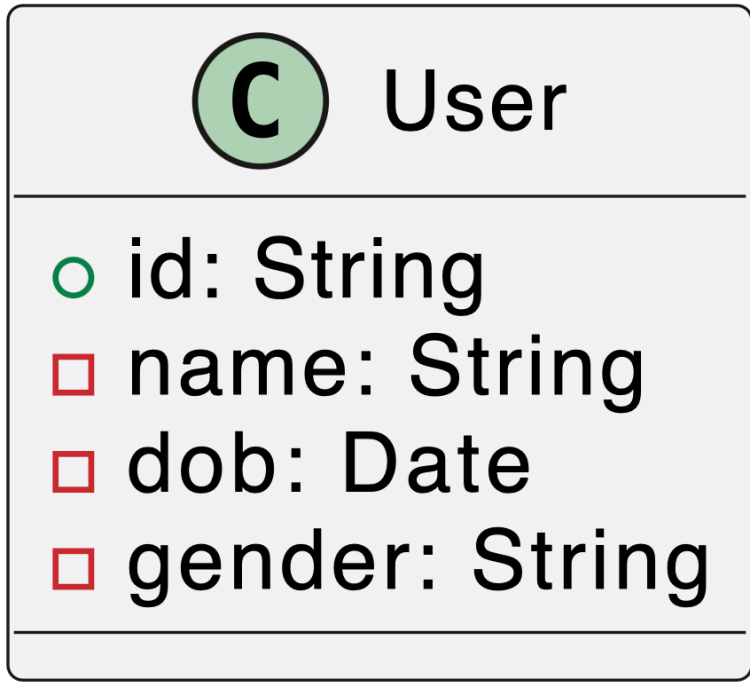
Encapsulation Smell – Deficient Encapsulation

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)

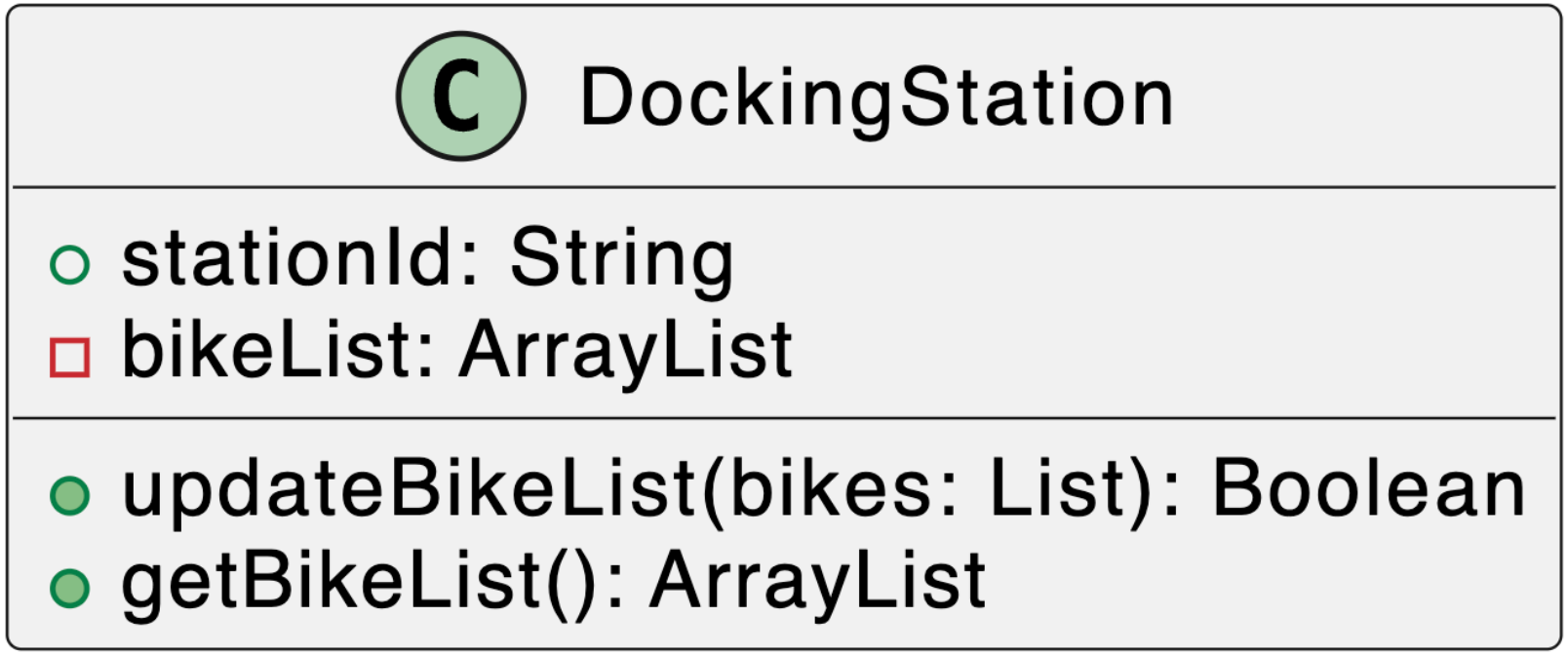
Rationale: Exposing details can lead to undesirable coupling. Each change in abstraction can cause change in dependent members

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

Impact: Affects changeability, extensibility, reliability,…

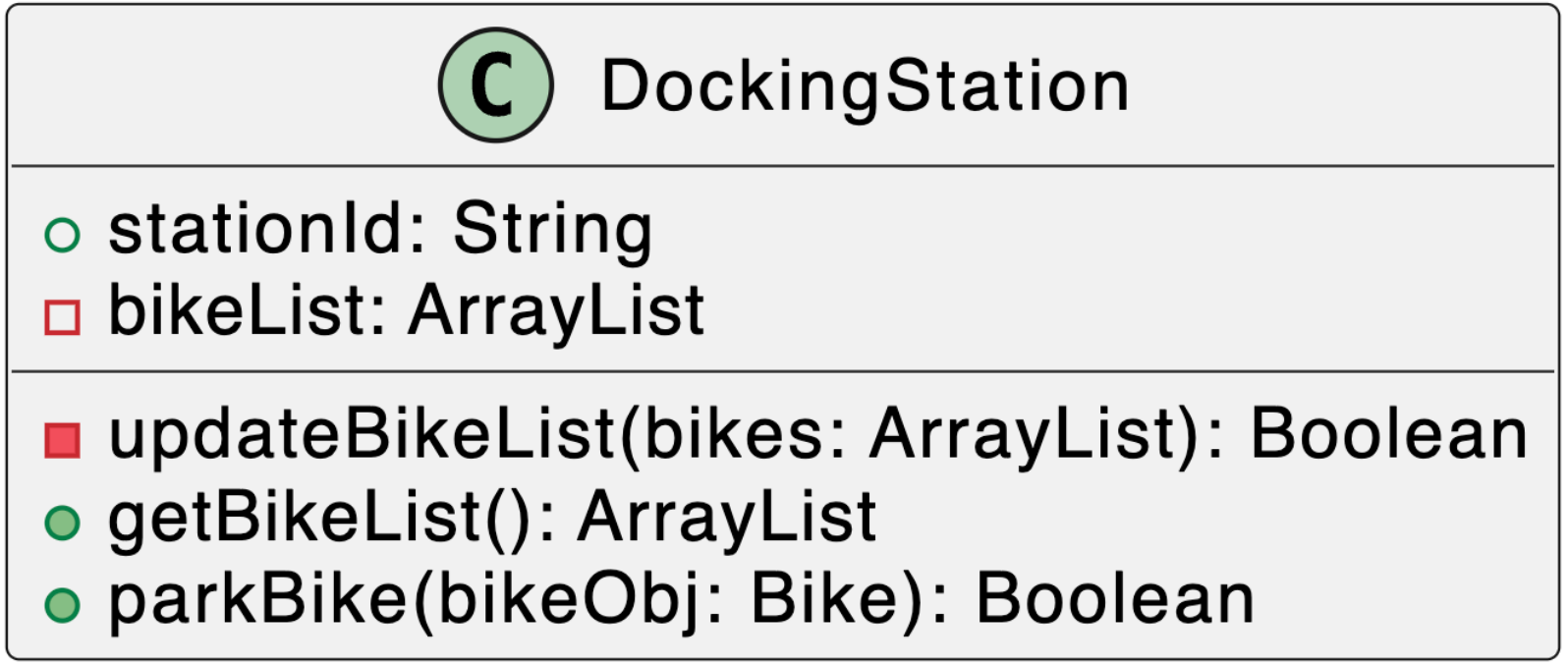
Encapsulation Smells – Leaky Encapsulations

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



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

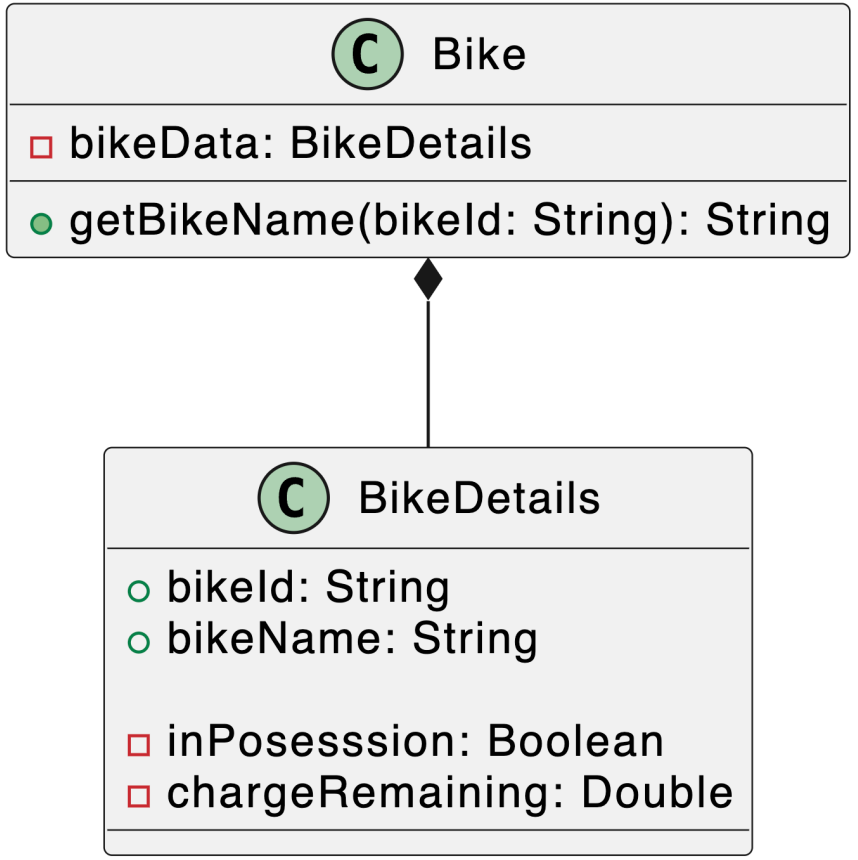
Encapsulation - Enablers

- 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

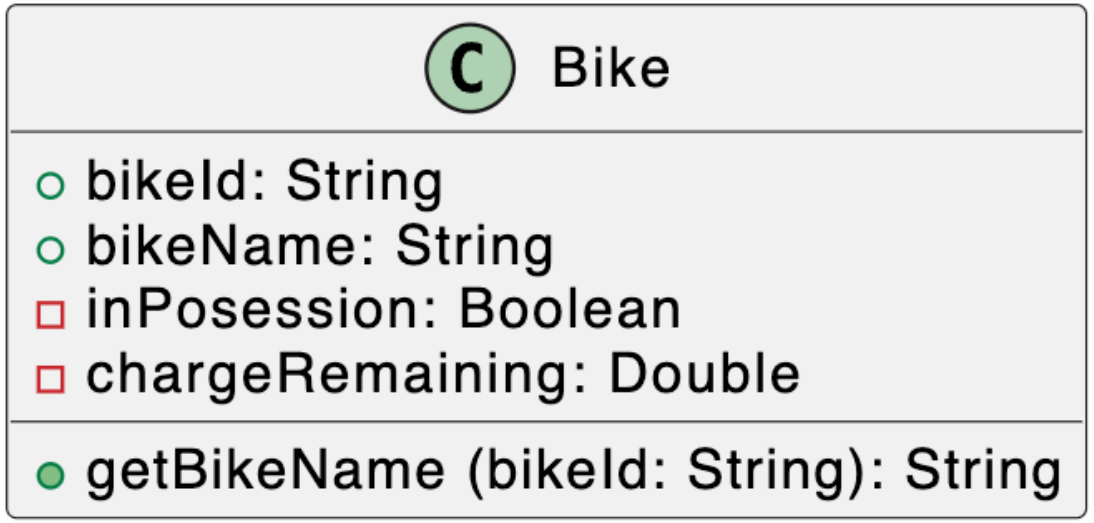
Modularization Smells – Broken Modularization

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

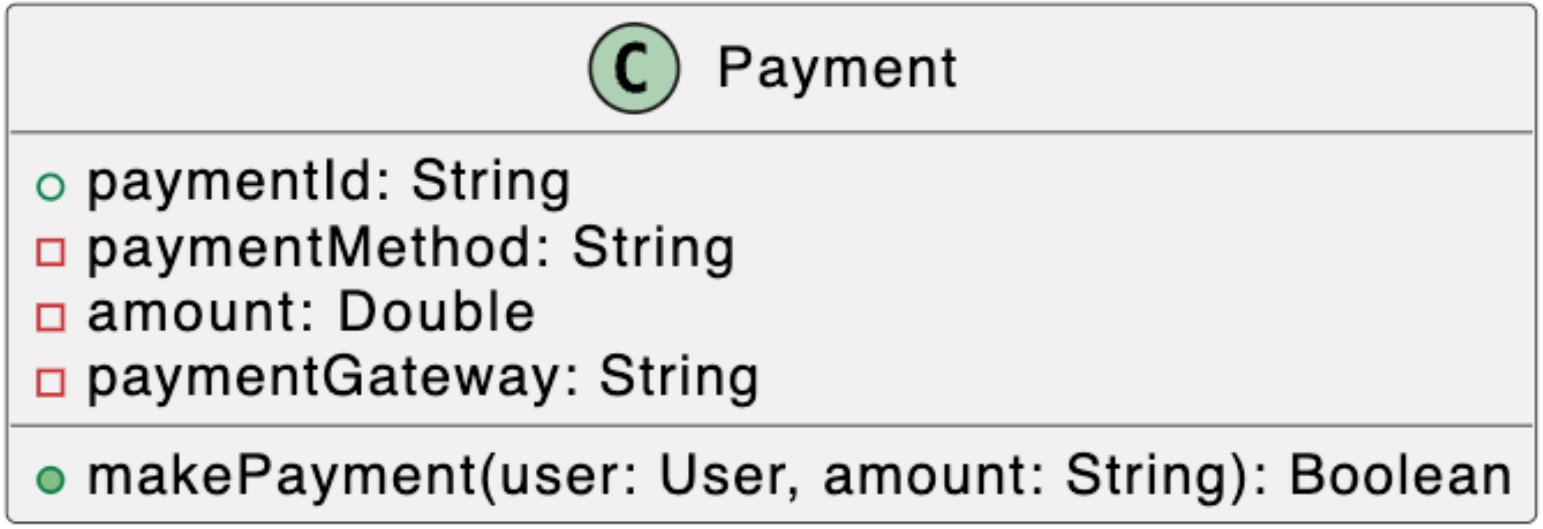
Impact: Affects changeability and extensibility, reusability, Reliability

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

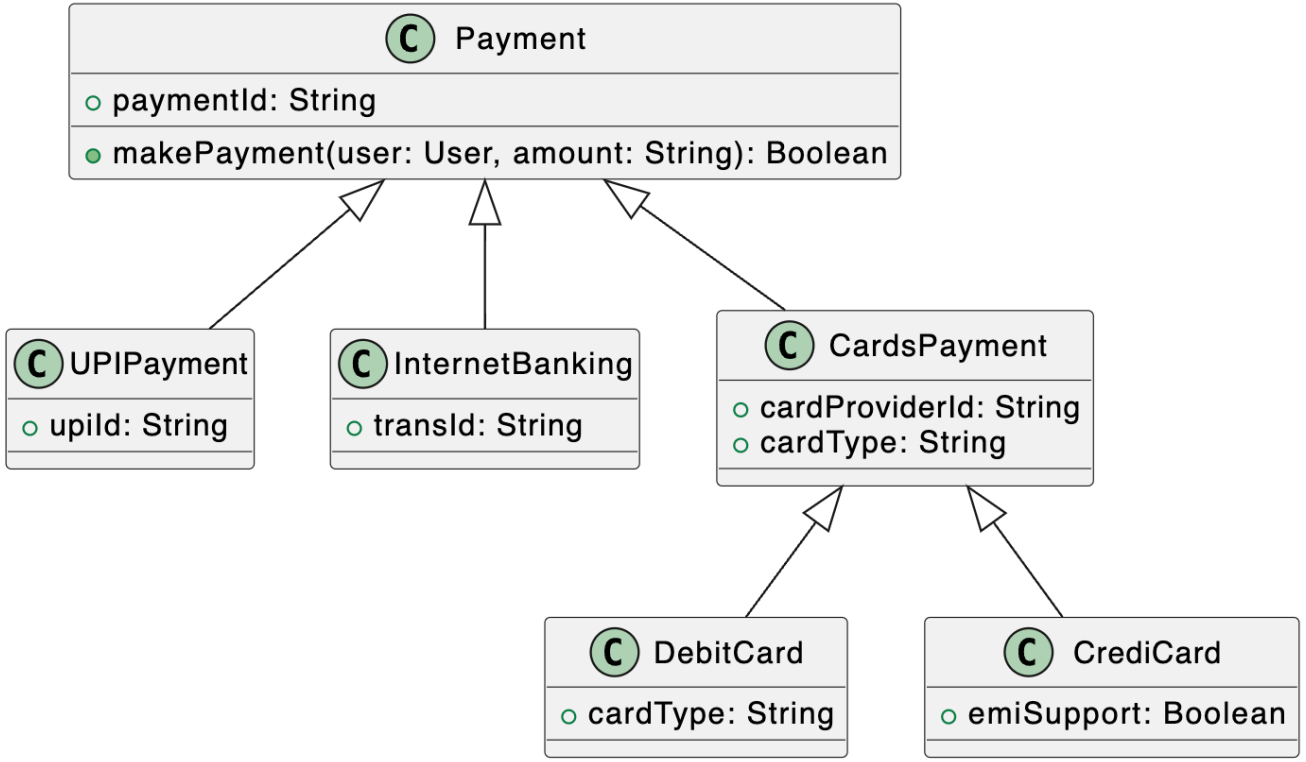
Scenario: In the e-vehicle scenario, user can pay in any mode of payment



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

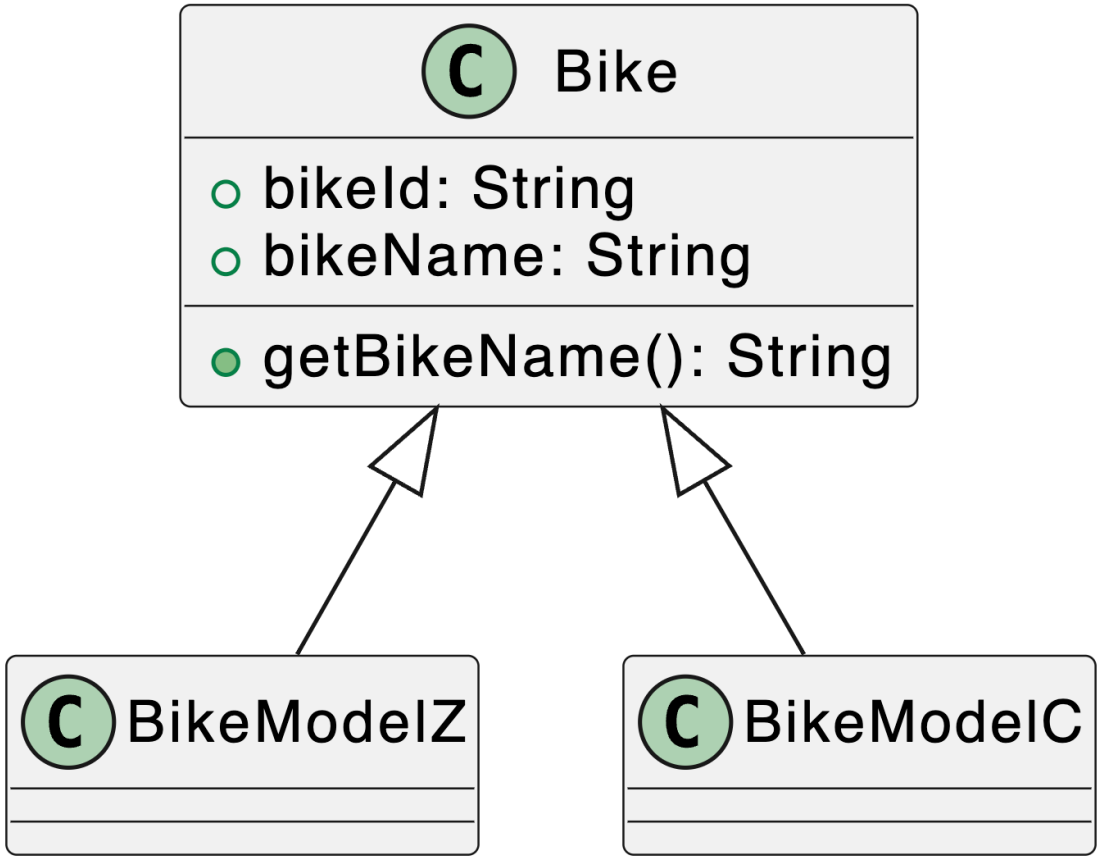
Rationale: Using chained if-else or Switch indicates issues with handling variations. Commonality among the types can also be used

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

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

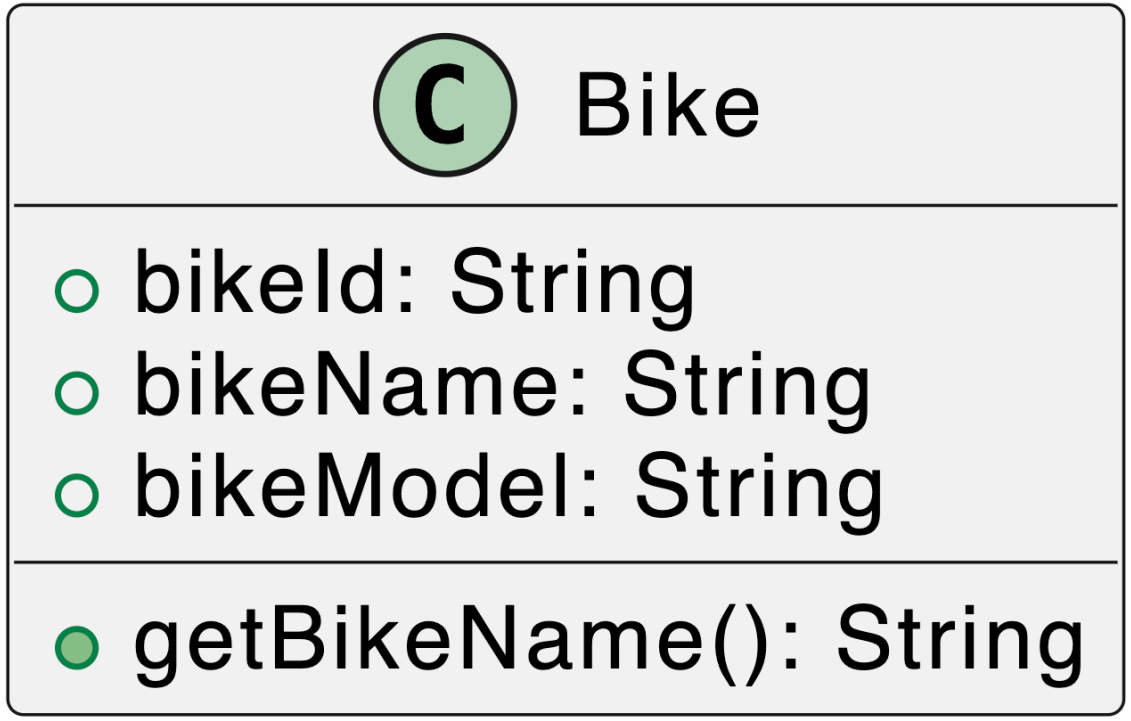
Hierarchy smells – Example Scenario

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!!

Group Activity

Thank You

Course website: karthikv1392.github.io/cs6401_se

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