Chpt 12: Information Hiding and Abstraction

Team 4 (TDD)

Table of Contents

Information Hiding and Abstraction

Modified Version

Information Hiding and Abstraction

- Definition (What) & Why
  - Abstraction or Information Hiding
- How can it go wrong
  - What causes “Big Balls of Mud”
    - Organizational and Cultural Problems
    - Technical Problems and Problems of Design
    - Fear of Over-Engineering
- How to properly do abstraction
  - Improving Abstraction Through Testing
  - Power of Abstraction
  - Leaky Abstractions
  - Picking Appropriate Abstractions
  - Abstractions from the Problem Domain
  - Abstract Accidental Complexity
  - Isolate Third-Party Systems and Code
  - Always Prefer to Hide Information

abstract memes

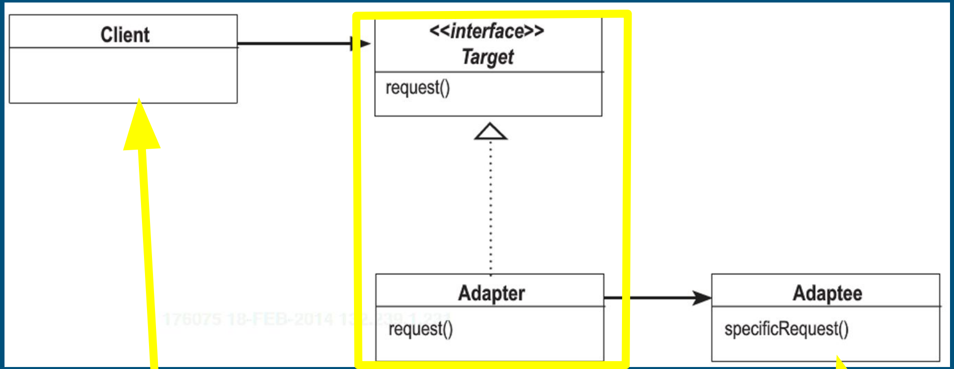
THINKING

Beyond "funny pictures", memes are abstractions. They capture the essence of an idea in a memorable, relatable, or comical way, and are often related to cultural or current events. Memes work because they share information in an efficient way across a culture.

Definition

- Abstraction == Information Hiding
  - “I don’t think the difference between the two is enough to concern us” - Farley
- Note that “abstract” can mean two opposite things
  - When we say something is “abstract”, it can mean that it’s conceptual-only and difficult to understand
- The abstraction we are talking about here, is to “abstract away” details to make a code base easier to understand or maintain
  - “draw lines in our code so that when we look at these lines from the ‘outside’ we don’t care about the details behind them” - Farley
- hiding code behavior from the consumers of a piece of code (black box)

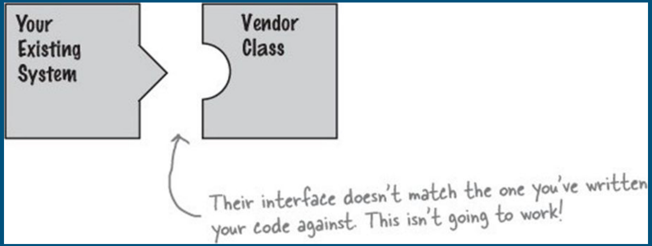
Adapter Pattern

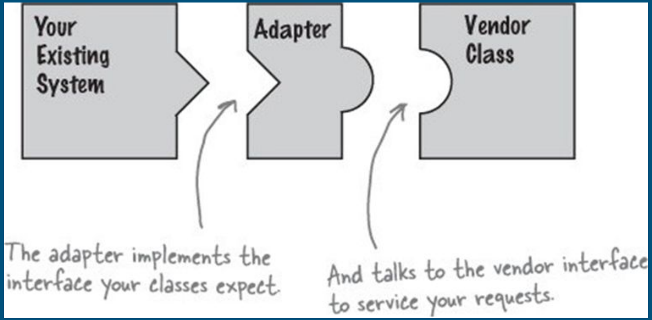


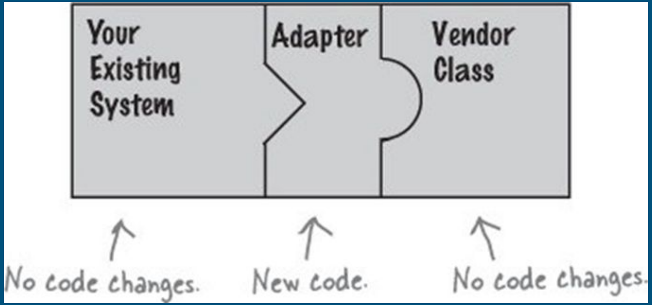
your code (create an instance of adapter obj)

your adapter class/interface

e.g. Mastodon API


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.



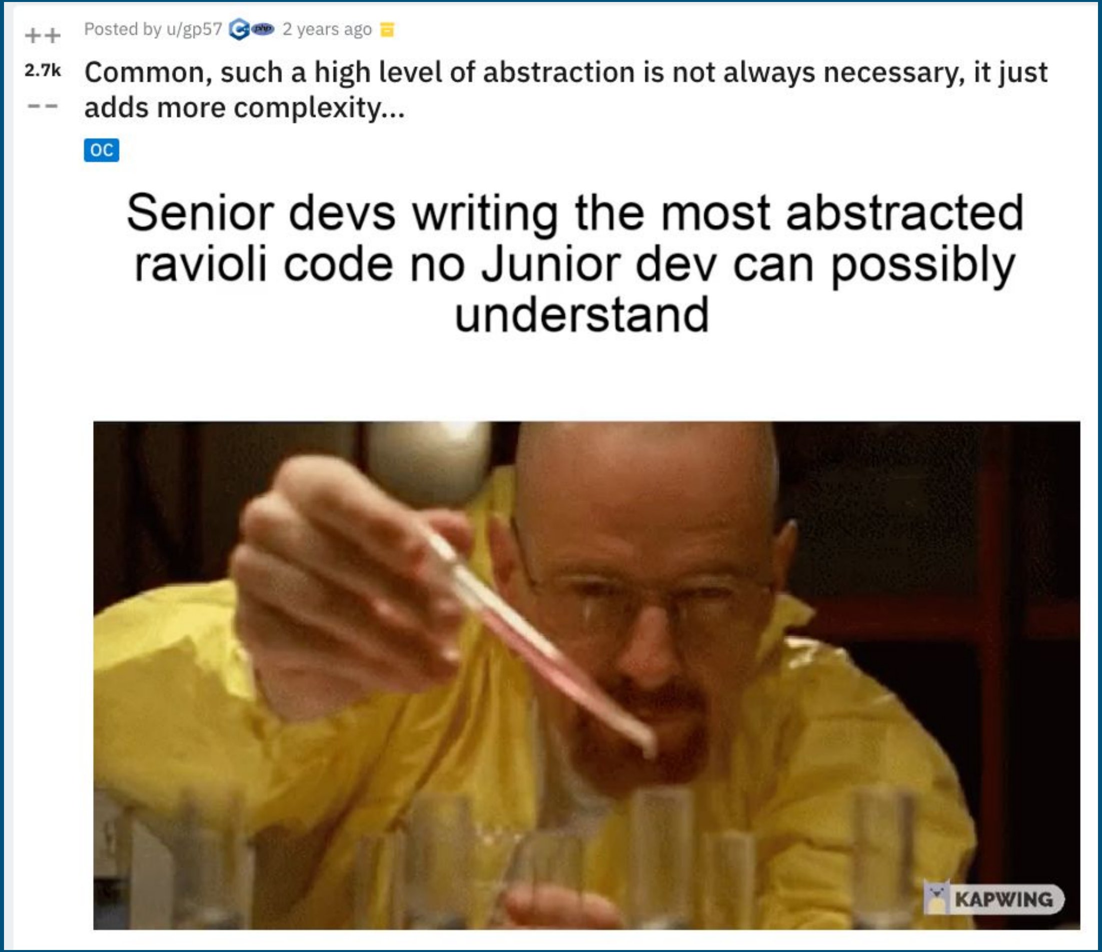
Why?

Abstraction helps managing complexity

- “We’d like to be able to focus on the work/code in front of us without worrying about what is going on elsewhere”

Table of Content

Information Hiding and Abstraction



++ Posted by u/gp57 2 years ago

2.7k Common, such a high level of abstraction is not always necessary, it just adds more complexity...

--

OC

Senior devs writing the most abstracted ravioli code no Junior dev can possibly understand

How Can it Go Wrong

- Farley explains how and why abstraction can be done poorly and cause “Big Balls of Mud”
    - Tangled and overly-complex code base that are difficult to work on

Interesting Youtube Video: Object-Oriented Programming is Bad (and most people are doing OOP wrong)

1. Organizational and Cultural Problems
2. Technical Problems and Problems of Design
3. Fear of Over-Engineering

Organizational and Cultural Problems

- Problem
  - Farley often hear complains about “my manager doesn’t let me do XXX” where XXX is either refactor/test/design better/fix a bug
  - If that’s really the case, you better find a new team
  - Most of the time, it’s something else
- Real problem (cause): People are often seeing coding as their only **duty of care**
  - We need to own the responsibility for **the quality of the code** we work on
  - This is in the interest of our employers, our users, and ourselves
- Supporting Claim: There is **no trade-off between speed and quality**
  - “If you are dropping **testing**, avoiding **refactoring**, or not taking time to find **more modular, more cohesive designs** to achieve some short-term delivery target**, you are going more slowly, not faster”**
  - We want “better software faster”, not “worst software faster”
- Solution
  - know what your jobs are (not just coding) and account for them in your estimates

Story Time

- The unambitious “risk-averse” noob manager - Amir
    - Just onboarded
    - Knew very little about the complex codebase
        - Impossible to know a lot b/c no comments (including javadoc) in our code
    - Scrum master for 2 teams (features)
    - Always asking “dumb” questions
- What did he do
    - Gave extra story points for every single story
    - Argued with PM to cut down features (skip some nice-to-haves)
- Finished early and had bandwidth to help out other teams (it was 2020, 1st release during the pandemic)
    - Before Farley: Control the scope & defensive planning
    - New Perspective: Helped me claim my duty of care (testing & refactoring)

Organizational and Cultural Problems

How does this link back to abstraction again…?

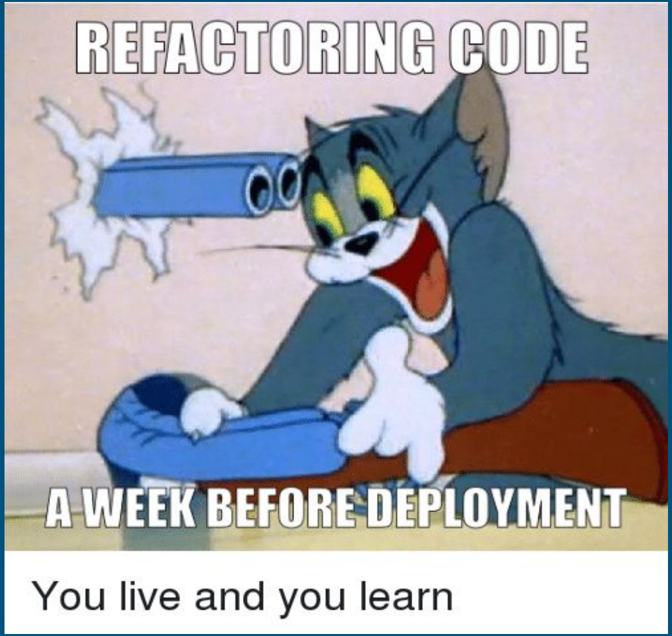
The premise of abstraction is that we understand SE is not just coding, but also things like **testing**, **refactoring**, and **modular/cohesive designs**

Otherwise we would just create broken abstractions and make the project “abstract”

Next, how do we make sure we are doing a good job? For example, refactoring?

Technical Problems and Problems of Design

Problem: “Many organizations are either afraid to change their code or have some kind of reverence for it that belies the reality. ”



Technical Problems and Problems of Design

Problem: “Many organizations are either afraid to change their code or have some kind of reverence for it that belies the reality. ”

- Farley argues the reverse: “if you can't or won't change the code, then the code is effectively dead.”
- Just as Fred Brooks writes in "Mythical Man Month," "As soon as one freezes a design, it becomes obsolete."

Solution: **Iteration, Incrementalism**, **Testing**

Fear of Over-Engineering

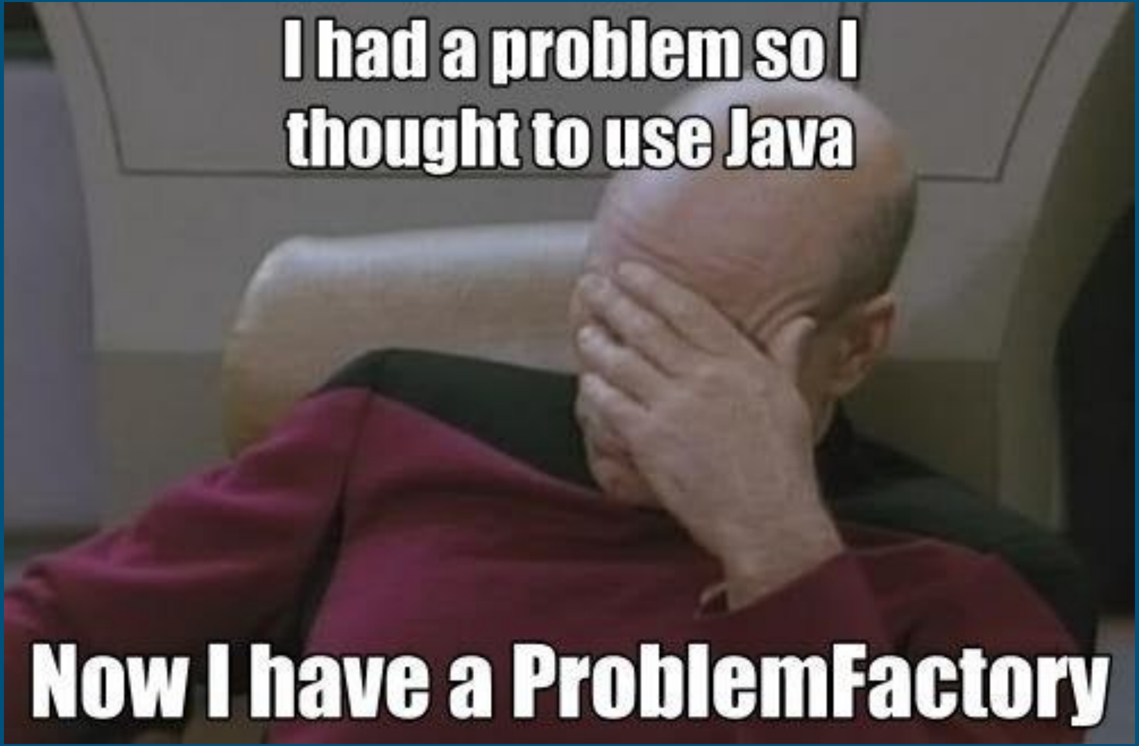
Problem: Almost the opposite of fear of refactoring, is rabbit hole of over-engineering

Solutions (more like arguments):

- Guard against chasing "technically shiny ideas."
  - We should always think of the simplest route to success, not the coolest or the one with the most technology that we can add to our CVs or resumes (RDD - Resume Driven Development)

- **Guard against making our solution "future-proof"**
  - The lambo problem
  - If you have ever said or thought, "We may not need this now, but we probably will in the future," then you were "future-proofing" it.
  - We should write code to solve the problem that faces us right now and only that.

Over-using design patterns


I had a problem so I thought to use Java
Now I have a ProblemFactory

Modified Version

Information Hiding and Abstraction

- Definition (What) & Why
  - Abstraction or Information Hiding
- Premise of good abstraction
  - How to avoid “Big Balls of Mud”
    - Know SE and clear duty of care
    - Ability to refactor (as you learn)
    - No over-Engineering
- How to properly do abstraction
  - Improving Abstraction Through Testing
  - Power of Abstraction
  - Leaky Abstractions
  - Picking Appropriate Abstractions
  - Abstractions from the Problem Domain
  - Abstract Accidental Complexity
  - Isolate Third-Party Systems and Code
  - Always Prefer to Hide Information

Improving Abstraction Through Testing

Overview

1. Automated regression testing enables frequent code changes at low cost by quickly revealing breaks. This supports a flat cost of change curve.
2. Writing tests first forces focusing on clear, simple external interfaces to make tests easy to express. This drives good abstraction design.
3. Tests act as "mini specifications" describing desired external behavior. This separates external requirements from internal implementation.

In summary, automated regression testing reduces the cost of changes, while test-first development creates natural abstraction boundaries by focusing design on external interfaces. Tests specify externally-visible behavior separately from implementation.

Power of Abstraction

Amazon Web Service’s S3

An API like that of Amazon Web Service’s S3 is deceptively simple.

A sequence of bytes can be submitted along with a label that can be used to retrieve it and the name of a “bucket” to place it in.

AWS will distribute it to data centers around the world and make it available to anyone who is allowed to access it and provide service-level agreements that will ensure that access in all but the most catastrophic of events is preserved.

This is abstracting some fairly complex stuff!

Power of Abstraction

Another example

- Data formats like HTML, XML, and JSON abstract away implementation details by providing a consistent, self-describing structure for communicating data between systems.
- While not the most efficient, they are popular because their text-based nature builds on the powerful "plain text" abstraction - the idea that information can be represented as streams of characters without worrying about lower-level details.
- "Plain text" itself is an abstraction that hides character encoding, byte ordering, and other complexities from developers.

The overall point is that abstractions are not inherent but purposefully designed organizing principles that allow managing complexity by hiding details. Good abstractions like "plain text" have evolved to become widely adopted conventions. Building on strong foundations like this enables creating higher-level abstractions.
So abstractions are artificial but immensely powerful tools essential for managing complexity through all levels of computing.

Leaky Abstractions

Definition

Leaky abstractions are defined as “an abstraction that leaks details that it is supposed to abstract away.”

This idea was popularized by Joel Spolsky, who went on to say this: *All non-trivial abstractions are leaky.*

The idea of “leaky abstractions” is not an argument against them; rather, it describes that abstractions are complex things that we need to take care over.

Leaky Abstractions

Types of leaks:

The two kinds of leaks discussed are:

1. Unavoidable leaks where abstractions do not perfectly model reality. These need to be understood and designed around.
2. Leaks due to poor design where the abstraction breaks down under certain conditions. These can be fixed by improving the design.

Picking Appropriate Abstractions

Definition

The nature of the abstractions that we choose matters. There is no universal “truth” here; these are models.

A good example of this are maps (the human kind of “map,” not the computer language data structure). All maps are abstractions of the real world, but we have different types of abstraction depending on our needs.

North Pole
New York
London
Madrid

Picking Appropriate Abstractions

Another Example

London's Underground Train Network Map by Harry Beck

Abstractions from the Problem Domain

Definition

- Abstractions from the Problem Domain are defined as "modeling and analyzing the problem domain to identify key concepts, behaviors, and bounded contexts that can inform the design."
- This allows for a natural separation of concerns, loose coupling between components, and a design that matches the domain rather than just technical considerations.
- Focusing on the problem domain leads to designs that match the domain better than just technical considerations.

Abstract Accidental Complexity

Definition

- Abstract Accidental Complexity is defined as "the technical constraints and details arising from the computing environment that inevitably leak into and affect the essential domain logic of a software system."
- Accidental complexity arises from technical constraints like hardware, databases, networks etc. These details inevitably leak into the essential domain logic.
- Creating clean abstractions for things like storage helps separate concerns and make code more modular and testable. But abstractions still leak, so handle failures gracefully.

Isolate Third-Party Systems and Code

Definition

- Isolating Third-Party Code is defined as "accessing third-party libraries and systems only through facades and adapters that abstract away implementation details." This provides flexibility to swap implementations without changing application code.
- Isolating third-party code makes your system more testable, flexible and composable. You can swap implementations without changing core logic.
- However, all abstractions are leaky to some degree. Third-party dependencies can still impose constraints that leak through the abstraction boundary. Careful interface design is needed to minimize coupling.

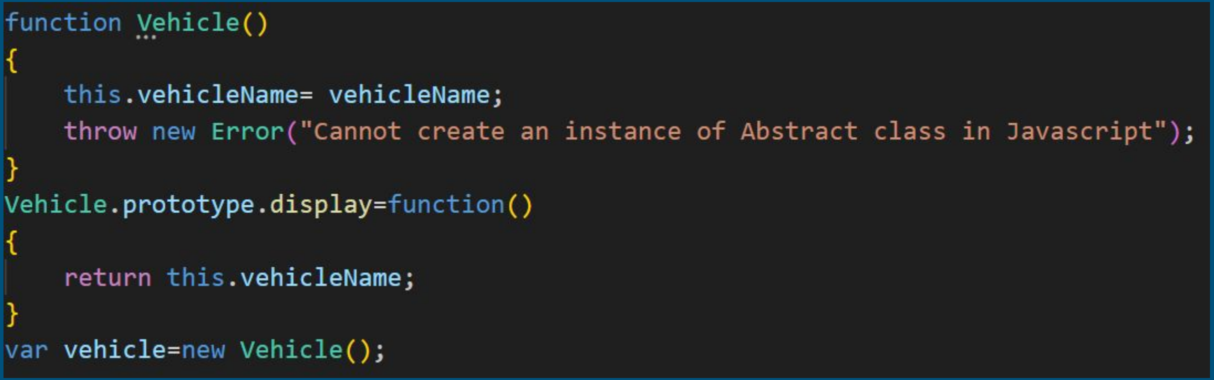
Always Prefer to Hide Information

Definition

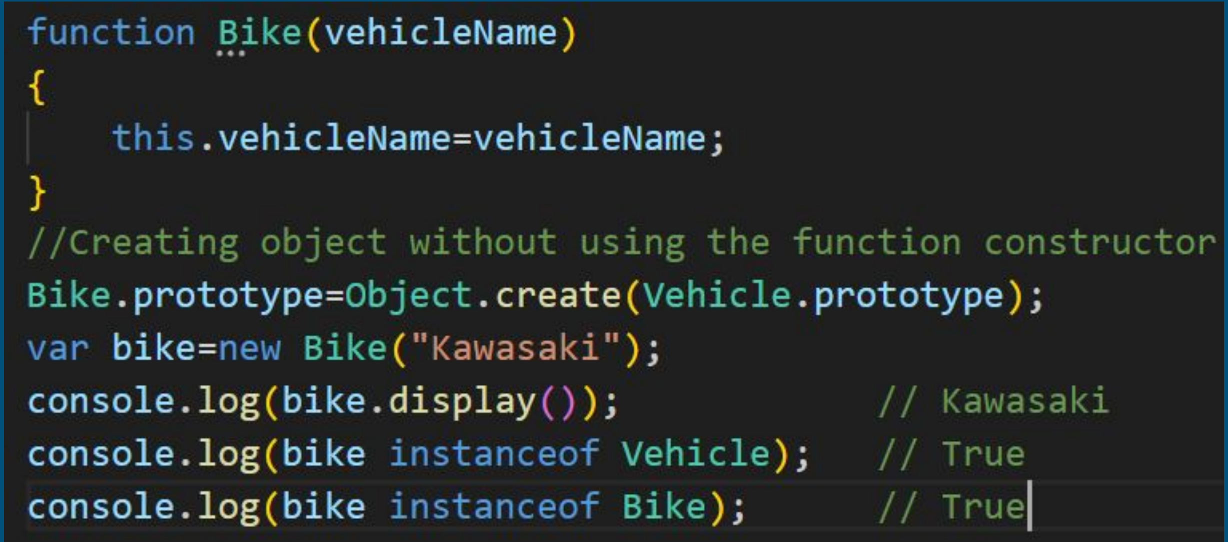
- Hiding Information is defined as "preferring generic representations in interfaces to avoid coupling to specifics, while remaining meaningful and helpful."
- The goal is finding the right level of abstraction that maintains the essence but hides unnecessary details, keeping options open for future change.
- The idea is to find the right level of abstraction that maintains the essence but avoids coupling to specifics.

Code examples

Abstraction in JS



```
function Vehicle()
{
    this.vehicleName= vehicleName;
    throw new Error("Cannot create an instance of Abstract class in Javascript");
}
Vehicle.prototype.display=function()
{
    return this.vehicleName;
}
var vehicle=new Vehicle();
```



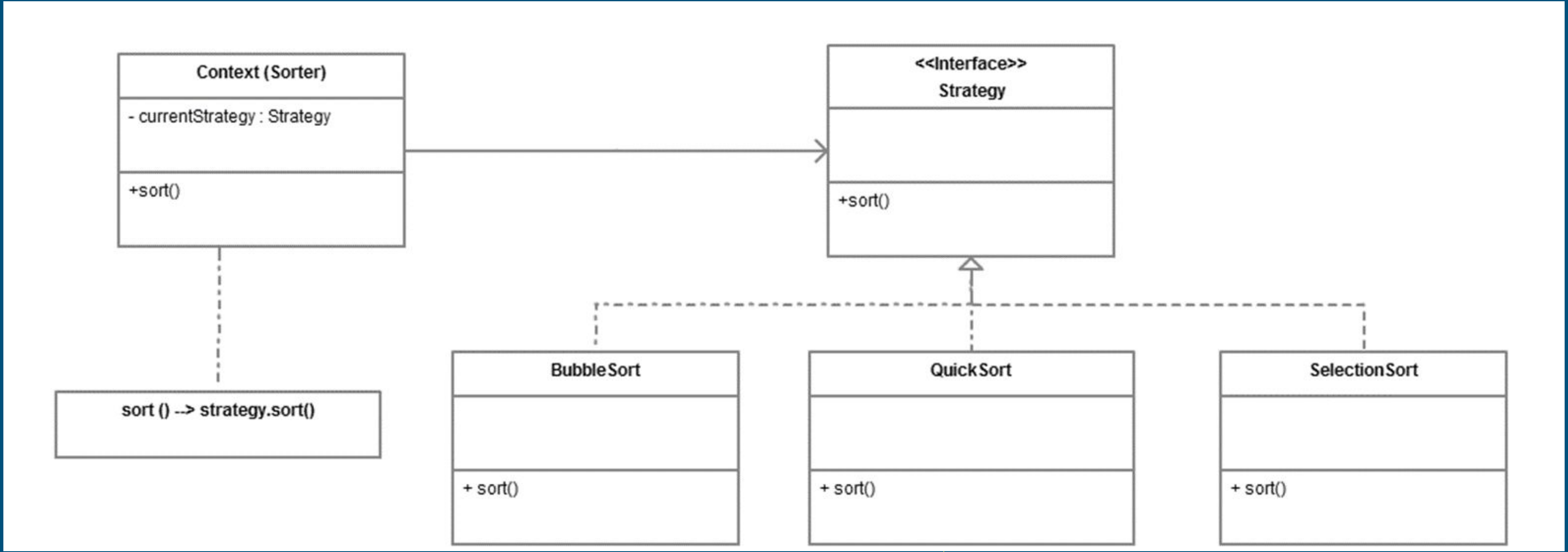
```
function Bike(vehicleName)
{
    this.vehicleName=vehicleName;
}
//Creating object without using the function constructor
Bike.prototype=Object.create(Vehicle.prototype);
var bike=new Bike("Kawasaki");
console.log(bike.display());              // Kawasaki
console.log(bike instanceof Vehicle);    // True
console.log(bike instanceof Bike);       // True
```

OCP (Open-closed Principle)

Open for extension
Closed for modification

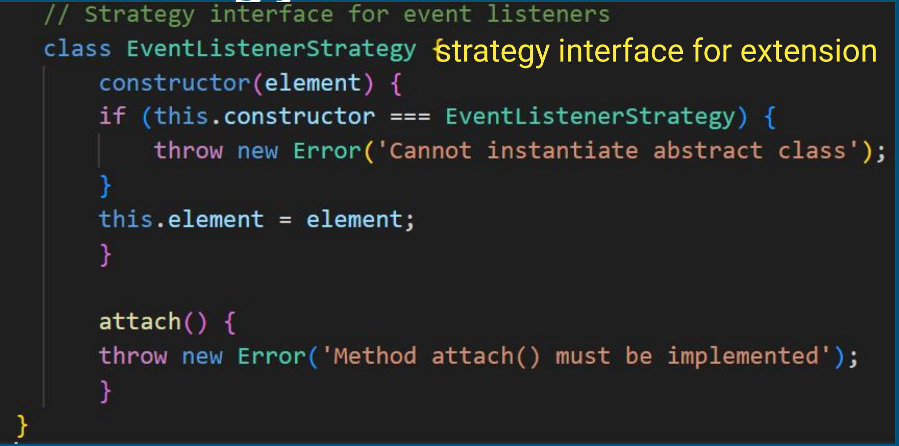
INFORMATION HIDING

Strategy Pattern



different strategies (open for extension)

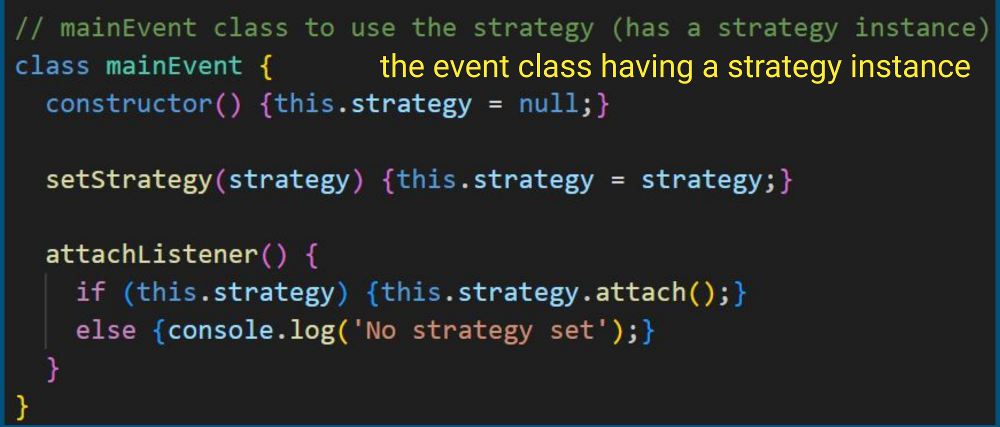
Strategy Pattern



```
// Strategy interface for event listeners
class EventListenerStrategy {
    constructor(element) {
    if (this.constructor === EventListenerStrategy) {
        throw new Error('Cannot instantiate abstract class');
    }
    this.element = element;
    }

    attach() {
    throw new Error('Method attach() must be implemented');
    }
}
```

strategy interface for extension

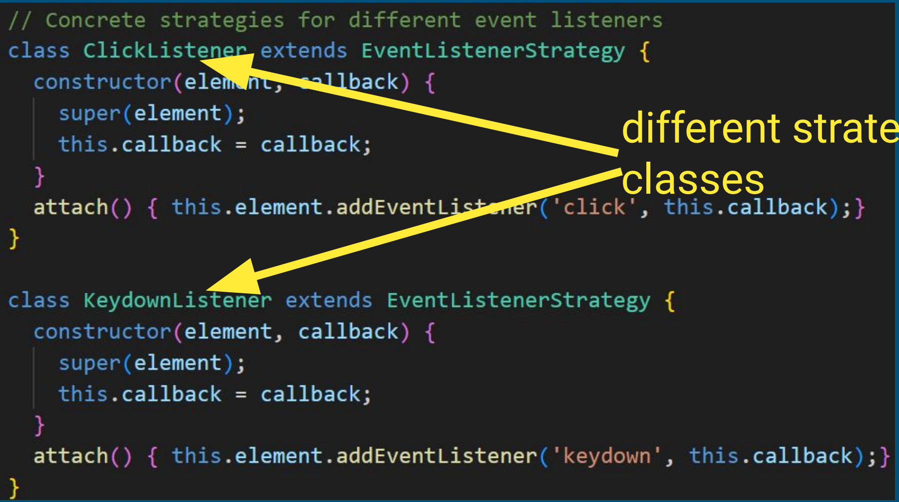


```
// mainEvent class to use the strategy (has a strategy instance)
class mainEvent {
  constructor() {this.strategy = null;}

  setStrategy(strategy) {this.strategy = strategy;}

  attachListener() {
    if (this.strategy) {this.strategy.attach();}
    else {console.log('No strategy set');}
  }
}
```

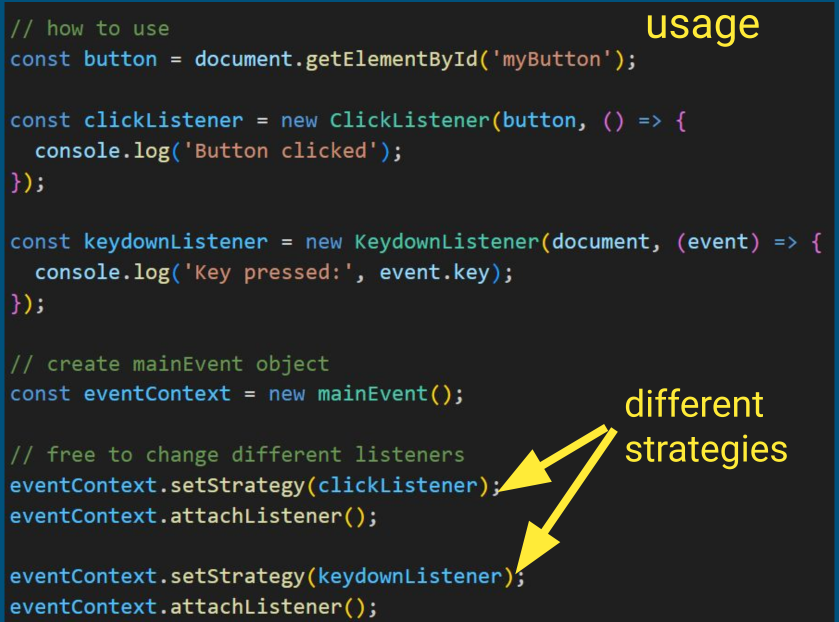
the event class having a strategy instance



```
// Concrete strategies for different event listeners
class ClickListener extends EventListenerStrategy {
  constructor(element, callback) {
    super(element);
    this.callback = callback;
  }
  attach() { this.element.addEventListener('click', this.callback);}
}

class KeydownListener extends EventListenerStrategy {
  constructor(element, callback) {
    super(element);
    this.callback = callback;
  }
  attach() { this.element.addEventListener('keydown', this.callback);}
}
```

different strategy classes



usage

```
// how to use
const button = document.getElementById('myButton');

const clickListener = new ClickListener(button, () => {
  console.log('Button clicked');
});

const keydownListener = new KeydownListener(document, (event) => {
  console.log('Key pressed:', event.key);
});

// create mainEvent object
const eventContext = new mainEvent();

// free to change different listeners
eventContext.setStrategy(clickListener);
eventContext.attachListener();

eventContext.setStrategy(keydownListener);
eventContext.attachListener();
```

different strategies

Thank you