Practical tips to refactor blocks of code based on clean coding principles

Books

1. Book 1: Good Code, Bad Code - Think Like a Software Engineer
2. Book 2: Clean Code in JS

General guidelines

1. Use **declarative** programming over imperative
2. Adopt and follow a **style guide**. Ex: A coding standard for **consistency**

Making code readable

Improve a name

Names of important units of code such as:

- Variables
- Functions
- Classes, properties, method, and objects

7 steps:

1. **Step 1:** Use descriptive names (i.e intention revealing names)
2. **Step 2:** Remove comments that substitute a descriptive name
3. **Step 3:** Use pronounceable names
4. **Step 4:** Do not unnecessarily truncate names (i.e avoid succinct but unreadable names)
5. **Step 5:** Do not use the variable's type in the name i.e avoid the Hungarian notation
6. **Step 6:** Describe side-effects of a function in its name
7. **Step 7:** Use the class context to name its properties and methods

```
// Step 1: Use descriptive names (i.e intention revealing names)
// -------
const x = 1; // Bad
const numberOfFruits = 1; // Good

const y = false; // Bad
const isANewsArticle = false; // Good
```

```
// Step 2: Remove comments that substitute a descriptive name
// -------
/* Represents a team */
class T { /* ... */ } // Bad

class Team { /* ... */ } // Good

/*
* @param n the player's name
* @return true is player is in the team
*/
f(n) {
  return pns.includes(n);
} // Bad

containsPlayer(playerName) {
  return playerNames.includes(playerName);
} // Good
```

```
// Step 3: Use pronounceable names
// -------
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
} // Bad

class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
} // Good
```

```
// Step 4: Do not unnecessarily truncate names i.e avoid succinct but unreadable names
// -------
```

```
function incId(id) { /* ... */ } // Bad
function incrementUserId(userId) // Good
```

```
// Step 5: Do not use the variable's type in the name (i.e avoid the Hungarian
notation)
// -------
const strName = '' // Bad
const decimalAmount = 3.2; // Bad
const fruitArray = []; // Bad
const userData = {} // Bad

const name = ''; // Good
const amount = 3.2; // Good
let fruits = [] // Good
const user = {} // Good
```

```
// Step 6: Describe side-effects of a function in its name
// -------
const getUserPreferences = (user) => {
  let userPreferences;
  // ...
  if (!userPreferences) {
    userPreferences = {}
  }
  return userPreferences
} // Bad

const setUserPreferencesIfNullAndGetIt = (user) => {
  let userPreferences;
  // ...
  if (!userPreferences) {
    userPreferences = {}
  }
  return userPreferences
} // Good
```

```
// Step 7: Use the class context to name its properties and methods
// -------
class TenancyAgreement {
  saveSignedDocument(
    tenancyAgreementSignedDocumentID,
    tenancyAgreementSignedDocumentTimestamp
  ) { /* ... */ }
} // Bad

class TenancyAgreement {
  // "TenancyAgreement" is the "namespace" for the names that exist inside it
  saveSignedDocument(
    documentID,
    documentTimestamp
  ) { /* ... */ }
} // Good
```

Improve a comment

3 steps:

1. **Step 1:** Comments cannot substitute readable code
2. **Step 2:** Comments should explain the "Why"
3. **Step 3:** Comments are good for high level summaries (Ex: Business logic, what a class does, etc)

```
// Step 1:
// -------
function generateId(data) {
  // data[0] contains the user's first name, and data[1] contains the user's
  // last name. Produces an ID in the form "{first name}.{last name}".
  return data[0] + "." + data[1];
} // Bad

function generateId(data) {
  return getFirstName(data) + "." + getLastName(data);
} // Good
```

```
// Step 2: Comments should explain the "Why"
// -------
function getUserId() {
  if (signupVersion.isOlderThan("2.0")) {
    // Legacy users (who signed up before 2.0) were assigned IDs
    // based on their name. See SCAL-2344 for more info.
    return firstName.toLowerCase() + "." + lastName.toLowerCase();
  }

  // Newer users are assigned IDs based on their username
  return username;
} // Good
```

```
// Step 3: Comments are good for high level summaries (Ex: Business logic)
// -------
/**
 * Encapsulates details of a user of the streaming service.
 *
 * This does not access the database directly and is, instead,
 * constructed with values stored in memory. It may, therefore,
 * be out of sync with any changes made in the database after
 * the class is constructed.
 */
class User {
  ...
} // Good
```

Fix deeply nested code

4 steps:

1. **Step 1:** Restructure to minimize nesting
    1. Use *smaller functions* (Ex: convert many **conditionals** and **if-elses** to **functions**)
    2. Use *guard clauses* (A guard clause is a check of integrity preconditions used to avoid errors during execution)
2. **Step 2:** Avoid negative conditionals (as much as possible)
3. **Step 3 (x):** Remove conditionals (switch, if-else) using polymorphism
    1. **Generally, ignore this step as it uses inheritance instead of composition!**
    2. **Composition** is a more modular and flexible way of extending base classes instead of inheritance
4. **Step 4:** Remove conditionals (*switch*, *if-else*) using the strategy pattern

```
// Step 1: Restructure to minimize nesting (smaller functions, guard clauses)
// -------
if (platform.state === 'fetching' && isEmpty(cart)) { /* ... */ } // Bad

function shouldShowLoading(platform, cart) {
  return platform.state === 'fetching' && isEmpty(cart);
}
// ...
if (shouldShowLoading(platform, cart)) { /* ... */ } // Good


function getPayAmount() {
  let result;
  if (isDead){
    result = deadAmount();
  } else {
    if (isSeparated){
      result = separatedAmount();
    } else {
      if (isRetired){
        result = retiredAmount();
      } else{
        result = normalPayAmount();
      }
    }
  }
  return result;
} // Bad

function getPayAmount() {
  if (isDead) return deadAmount(); // Guard clause
  if (isSeparated) return separatedAmount(); // Guard clause
  if (isRetired) return retiredAmount(); // Guard clause
  return normalPayAmount();
} // Good
```

```
// Step 2: Avoid negative conditionals (as much as possible)
// -------
```

```
if (!isEmployed) { } // Bad
if (!isDomNodeNotPresent(node)) {} // Bad

if (isEmployed) { } // Good (less cognitive load)
if (isDomNodePresent(node)) {} // Good (less cognitive load)
```

```
// Step 3 (x): Remove conditionals using polymorphism
// -----------
// Only use if inheritance is a must! (i.e genuine "is-a" relationships)
// Else, go for strategy pattern (or composition)
class Airplane {
  // ...
  getCruisingAltitude() {
    switch (this.type) {
      case "Boeing 777":
        return this.getMaxAltitude() - this.getPassengerCount();
      case "Air Force One":
        return this.getMaxAltitude();
      case "Cessna":
        return this.getMaxAltitude() - this.getFuelExpenditure();
    }
  }
} // Bad

class Airplane { /* ... */ }
class Boeing777 extends Airplane {
  getCruisingAltitude() {
    return this.getMaxAltitude() - this.getPassengerCount();
  }
}
class AirForceOne extends Airplane {
  getCruisingAltitude() {
    return this.getMaxAltitude();
  }
}
class Cessna extends Airplane {
  getCruisingAltitude() {
    return this.getMaxAltitude() - this.getFuelExpenditure();
  }
} // Good
```

```
// Step 4: Remove conditionals using the strategy pattern
// -------
class Airplane {
  // ...
  getCruisingAltitude() {
    switch (this.type) {
      case "Boeing 777":
        return this.getMaxAltitude() - this.getPassengerCount();
      case "Air Force One":
        return this.getMaxAltitude();
      case "Cessna":
        return this.getMaxAltitude() - this.getFuelExpenditure();
      }
    }
} // Bad

class Airplane {
  constructor (type) {
    this.type = type;
  }

  getCruisingAltitudeStrategies() {
    return {
      boeing777: () => this.getMaxAltitude() - this.getPassengerCount(),
      airforceone: () => this.getMaxAltitude(),
      cessna: () => this.getMaxAltitude() - this.getFuelExpenditure()
    }
  }

  getCruisingAltitude() {
    const airplaneStrategyKey = this.type.split(" ").map(word =>
word.toLowerCase()).join("");
    return this.getCruisingAltitudeStrategies()[airplaneStrategyKey];
  }
} // Good


function logMessage(message = "CRITICAL::The system ..."){
  const parts = message.split("::");
  const level = parts[0];

  switch (level) {
    case 'NOTICE':
      console.log("Notice")
      break;
    case 'CRITICAL':
      console.log("Critical");
      break;
    case 'CATASTROPHE':
      console.log("Castastrophe");
      break;
  }
} // Bad

const strategies = {
  criticalStrategy: (param) => console.log("Critical: " + param),
  noticeStrategy: (param) => console.log("Notice: " + param),
  catastropheStrategy: (param) => console.log("Catastrophe: " + param)
}
function logMessage(message = "CRITICAL::The system ...") {
  const [level, messageLog] = message.split("::");
  const strategy = `${level.toLowerCase()}Strategy`;
  const output = strategies[strategy](messageLog);
} // Good
```

Fix loops

1 step:

1. **Step 1:** Favor **functional programming** instead of iterative programming i.e use more declarative methods
   1. Only use iterative loops if built-in iterator methods are insufficient

```
// Step 1: Favour functional programming instead of iterative programming
// i.e use more declarative methods
// -------
function getUnpaidInvoices(invoiceProvider) {
  const unpaidInvoices = [];
  const invoices = invoiceProvider.getInvoices();
  for (var i = 0; i < invoices.length; i++) {
    if (!invoices[i].isPaid) {
      unpaidInvoices.push(invoices[i]);
    }
  }
  return unpaidInvoices;
} // Bad

function getUnpaidInvoices(invoiceProvider) {
  return invoiceProvider.getInvoices().filter(invoice => {
    return !invoice.isPaid;
  });
} // Good
```

Improve functions

9 steps:

1. **Step 1:** Reduce the number of arguments
2. **Step 2:** Use named arguments i.e object destructing in JavaScript
3. **Step 3:** Use descriptive types for arguments (TypeScript example)
   1. Either create and use a **class** for a specific type or use an **enum**
4. **Step 4:** Use default arguments instead of short-circuiting
5. **Step 5:** Remove boolean flags from input parameters
   1. *Boolean flags in parameters indicate the function has two paths, increasing complexity!*
   2. **Replace with a function** of its own
6. **Step 6:** Avoid using unexplained values (ex: Magic numbers)
   1. Replace with a **well-named constant**
   2. Replace with a **well-named function**
7. **Step 7:** Use anonymous functions for small things
8. **Step 8:** Make functions follow the **Single Responsibility Principle (SRP)**
   1. **Break a function** doing too many things **into smaller functions**.
      1. This helps a function focus on one concept rather than involving subproblems in the lower levels.

2. *There are ways to break up functions of higher arity and supply arguments as and when needed instead of knowing everything beforehand.* We can do this with:
    1. Use **partial application** (A higher order function, H.O.F)
    2. Using **currying** helps (A higher order function, H.O.F)
        1. *Bonus:* Use currying with functions that use **array methods** for the **return** value. This prevents you from needing to know all the *parameters for a list filter or map or find* beforehand!

9. **Step 9:** Make functions stay within a **single layer of abstraction** (Note: Partial application & currying help)

```
// Step 1: Reduce the number of arguments
// -------
function printUserDetails(id, firstName, lastName, age, gender) { /* ... */ } // Bad
function printUserDetails(user) { /* ... */ } // Good

// Use only those params that a function needs:
function printUserAge(user) { /* ... */ } // Bad
function printUserAge(age) { /* ... */ } // Good ( Ex: printUserAge(user.age); )
```

```
// Step 2: Use named arguments i.e object destructuring in JavaScript
// -------
function sendMessage(message, priority, allowRetry) { /* ... */ };
sendMessage("hello", 1, true); // Bad

function sendMessage({ message, priority, allowRetry }) { /* ... */ };
sendMessage({
  message: 'hello',
  priority: 1,
  allowRetry: true,
}); // Good
```

```
// Step 3: Use descriptive types for arguments (TypeScript example)
// -------
// T.S
function sendMessage(String message, number priority, boolean allowRetry) {
    // ...
} // Bad

class MessagePriority {
  // ...
  constructor(number priority) { ... }
} // A class to replace priority - number was too generic
enum RetryPolicy {
    ALLOW_RETRY,
    DISALLOW_RETRY
} // An enum to replace a true or false value since we might add more policies later
function sendMessage(String message, MessagePriority priority, RetryPolicy
retryPolicy) {
    // ...
} // Good
```

```
// Step 4: Use default arguments instead of short-circuiting
// -------
function setName(name) {
    const newName = name || 'Juan Palomo'; // A necessity in JavaScript before ES6
} // Bad

function setName(name = 'Juan Palomo') {
    // ...
} // Good
```

```
// Step 4: Remove boolean flags from input parameters
// -------
function bookTicket(customer, isPremium) {
  // ...
  if (isPremium) {
    premiumLogic(); // Path 1
  } else {
    regularLogic(); // Path 2 (Already doing too many things)
  }
} // Bad

function bookPremiumTicket(customer) {
  premiumLogic();
}
function bookRegularTicket(customer) {
  regularLogic();
} // Good
```

```
// Step 6: Avoid using unexplained values (ex: Magic numbers)
// -------
function getKineticEnergyInJoules() {
  return 0.5 * getMassKg() * 907.1847 * Math.pow(getSpeedMph() * 0.44704, 2);
} // Bad

// TS example:
class Vehicle {
  private const Double KILOGRAMS_PER_US_TON = 907.1847; // Constants in capital
(convention)
  private const Double METERS_PER_SECOND_PER_MPH = 0.44704;

    getKineticEnergyInJoules() {
      return 0.5 * getMassKg() *
      KILOGRAMS_PER_US_TON * Math.pow(
          getSpeedMph() * METERS_PER_SECOND_PER_MPH, 2 // well-named constants
        );
    }
} // Good

// Alternate: Use well-named functions
```

```
getKineticEnergyInJoules() {
  return 0.5 * getMassUsTon() *
    kilogramsPerUsTon() *
    Math.pow(getSpeedMph() * metersPerSecondPerMph(), 2);
}
```

```
// Step 7: Use anonymous functions for small things!
// (Note: Anonymous functions for big things is a readability and debuggability
concern!)
// -------
function getUsefulFeedback(allFeedback) {
  const usefulFeedback = [];
  for (let i = 0; i < allFeedback.length; i++) {
    const feedback = allFeedback[i];
    if (!feedback.getComment().isEmpty()) {
      usefulFeedback.push(feedback);
    }
  }
  return usefulFeedback;
} // Bad

function getUsefulFeedback(allFeedback) {
  return allFeedback.filter(feedback => !feedback.getComment().isEmpty()); //
Anonymous function
} // Good
```

```
// Step 8: Make functions follow Single Responsibility Principle (SRP)
// -------
// (a) Split the function into smaller functions of just one responsibility each
function calculateSalary(employee) {
  // Calculating salary(responsibility 1)
  let salary = employee.hoursWorked * employee.hourlyRate;2)
    let report = /*...*/;
  console.log(report); // Printing a report (responsibility 2)
  return salary;
} // Bad


function calculateSalary(employee) {
    return employee.hoursWorked * employee.hourlyRate;
}
function generateReport(employee, salary) {
    let report = /*...*/;
    console.log(report);
} // Good


// (b) Partial application (We don't need to know all arguments at the same time)
// A partially applied function reduces the total number of arguments for a function
(known as "arity") while giving you back a function that takes in fewer arguments.
function createEvent(location, owner, date, time) {
  return {
    ...location,
    owner,
    date,
    time
  }
} // Bad
createEvent("Bengaluru", "Brad", "10-07-20", "4pm")
createEvent("Bengaluru", "Bob", "10-11-20", "10am")
// Two problems:
// 1. Function needs to know a lot of things: location, owner, & time
// 2. Repetition


function createLocationSpecificEvent(location, owner) {
  return (date, time) => ({
    ...location,
    owner,
    date,
    time
  })
}
const createBengaluruEvent = createLocationSpecificEvent("Bengaluru", "Brad")
createBengaluruEvent("10-07-20", "4pm")
createBengaluruEvent("10-11-20", "10am") /
// Solved one problem:
// 1. Reusable, SRP function
// A problem that still persists:
// 1. Repetition (of createBengaluruEvent) - not exactly reusable! What if owner
changed?


// (c) Currying takes partial application to the next level (i.e highly flexible)
// It exactly one argument and returns a series of functions that each return a
function taking in one argument until it is fully resolved.
const createEvent = location => owner => date => time => ({
  ...location,
  owner,
  date,
  time
})

const bangaloreEvent = createEvent("Bangalore")
const createBradEvent = bangaloreEvent("Brad")
eventsUnderBrad("10-07-20")("4pm")
const mumbaiEvent = createEvent("mumbai") // Good!
// Super testable and more reusable!
// No repetition - completely reusable


// BONUS (c.a) Use currying with functions that use array methods for the return value
// i.e instead of iterative loops
// -------
```

```
const filterUsers = (users, field, value) => {
  return users.filter(user => user[field] === value)
} // Bad
// A highly restrictive function!
// Need to know everything before hand
// Adds to repetition

const filterListOfObjects = field => value => users => users.fitler(user => user[field
=== value])
const matchByAge = filterListOfObjects("age")
const matchAboveAge18 = matchByAge(18)
const usersAboveAge18 = matchAboveAge18(users) // Good
```

```
// Step 9: Make functions stay within a single layer of abstraction
// -------
function parseBetterJSAlternative(code) {
  const REGEXES = [
    // ...
  ];
  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      // ...
    });
  });
  const ast = [];
  tokens.forEach((token) => {
    // lex...
  });
  ast.forEach((node) => {
    // parse...
  });
} // Bad
// One function is responsible for tokenizing (level 1), lexing (level 2),
// parsing (level 3), ... In the hierarchy of code, these are 3 code levels, one above
the other.


const REGEXES = [ /* ... */ ];
function tokenize(code) { // Only focused on the tokenize level (level 1)
  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      tokens.push( /* ... */ );
    });
  });
  return tokens;
}
function lexer(tokens) { // Only focused on the lexing level (level 2)
  const ast = [];
  tokens.forEach((token) => ast.push( /* */ ));
  return ast;
}
function parseBetterJSAlternative(code) { // Only focused on the parsing level (level
3)
  const tokens = tokenize(code);
  const ast = lexer(tokens);
  ast.forEach((node) => // parse...);
} // Good
```

Avoid surprises

Fix a return statement

3 steps:

1. **Step 1:** Do not return magic values (Ex: returning `-1` to indicate a missing value from a list or array during a search)
    1. Replace with null return, error, or an optional (in typescript) (if it is available, Ex: TypeScript)
    2. *Note*: Returning magic values can happen buy accident too!
2. **Step 2:** Consider using the **Null Object pattern** for simple returns (Ex: Returning an empty list/collection instead of null)
    1. The *null object pattern* is an alternative to returning null (or an empty optional) when a value can't be obtained. The idea is that instead of returning null, a valid value is returned that will cause any downstream logic to behave in an innocuous way.
3. **Step 3:** Avoid complicated Null Objects

```
// Step 1: Do not return magic values - Replace with null return, error, or an
optional (TS)
// -------
class User {
  getAge() {
    // ...
    return -1; // If user does not have an age (Magic value) (Bad!)
  }
}
function getMeanAge(users) {
  if (users.isEmpty()) {
    return null;
  }
  let sumOfAges = 0;
  for (user in users) {
    sumOfAges += user.getAge();
  }
  return sumOfAges / users.length;
} // Bad
// getMeanAge() might return the wrong mean if a user has no age
// (a) Return null instead of a magic value (solution 1)
class User {
  getAge() { // In TS this would be: getAge(): number? { ... }
    return age; // age is NULL if not set
  }
};
function getMeanAge(users) {
  if (users.isEmpty()) {
    return null;
  }
  let sumOfAges = 0;
  for (user in users) {
    // Overhead: Need to account for null values
    // Still, it is better than magic values where the error of calculations is
suppresed
    if (user.getAge()) {
      sumOfAges += user.getAge();
    } else {
        throw new Error('Age missing for some users');
    }
  }
  return sumOfAges / users.length;
} // Bad
// getMeanAge() might return the wrong mean if a user has no age

// Note: Magic values can happen buy accident too
```

```
// Step 2: Consider using the Null Object pattern for simple returns
// -------
function getClassNames(element: HtmlElement): Set {
  const attribute: string? = element.getAttribute("class");
  if (attribute == null) {
    return new Set(); // Null object pattern
  }
  return new Set(attribute.split(" "));
}
// ...
function isElementHighlighted(element: HtmlElement) {
  return getClassNames(element).contains("highlighted"); // No need to check for a
null
} // Good
```

```
// Step 3: Avoid complicated Null Objects
// -------
class CoffeeMug {
  private constructor(diameter: number, height: number) { /* ... */ }
  getDiameter() { /* ... */ }
  getHeight() { /* ... */ }
};
class CoffeeMugInventory {
  mugs;
  // ... Constructs and returns a zero-size coffee mug when no mugs are available!
  //
  getRandomMug(): CoffeeMug {
    if (mugs.isEmpty()) {
      return new CoffeeMug(diameter: 0.0, height: 0.0);
    }
    return mugs[Math.floor(Math.random(0, mugs.size()))];
  }
}; // Bad

// The author of the codeno doubt has good intentions; they’re trying to make life
easier for callers of the getRandomMug() function by not forcing them to handle a
null. But unfortunately this creates a situation that is potentially surprising,
because a caller to the function is given the false impression that they will always
get a valid CoffeeMug back.

// It would probably be better to simply return null from getRandomMug() when there
are no mugs from which to select a random one. This makes it unmistakably obvious in
the code’s contract that the function may not return a valid mug and leaves no room
for surprise when this does in fact happen.
```

Fix side-effects

2 steps:

1. **Step 1:** *Do not fix* intentional side-effects (TS example)
2. **Step 2: Avoid** unintentional side-effects (or) make it **obvious**

```
// Step 1: Do not fix intentional side-effects (TS example)
// -------
class UserDisplay {
  private canvas: Canvas;
  // ...
  displayErrorMessage(message: string): void {
    canvas.drawText(message, Color.RED); // Expected side effect inside a display
method
  }
} // Good
```

```
// Step 2: Avoid unintentional side-effects (or) make it obvious
// -------
class UserDisplay {
  private canvas: Canvas;
  // ...
  getPixel(x: number, y: number): Color {
    canvas.redraw(); // Triggering a redraw event is an unintentional side effect!
    const data: PixelData = canvas.getPixel(x, y);
    return new Color(
      data.getRed(),
      data.getGreen(),
      data.getBlue()
    );
  }
} // Bad
// Not only is canvas.redraw() an unintentional side effect while getting a pixel,
// it can potentially be a very expensive operation (a perf bottelneck)!


// Solution
// (a) Avoid a side-effect if possible
class UserDisplay {
  private canvas: Canvas;
  // ...
  getPixel(x: number, y: number): Color {
    const data: PixelData = canvas.getPixel(x, y);
    return new Color(
      data.getRed(),
      data.getGreen(),
      data.getBlue()
    );
  }
} // Good
// NO canvas.redraw(); !

// (b) Make a side-effect "obvious" if it cannot be removed!
class UserDisplay {
  private canvas: Canvas;
  // ...
  Color redrawAndGetPixel(x: number, y: number) {
    // The function name makes it obvious to the caller that it redraws first
    canvas.redraw(); // Required side-effect?
    const data: PixelData = canvas.getPixel(x, y);
    return new Color(
      data.getRed(),
      data.getGreen(),
      data.getBlue()
    );
  }
} // Good
```

Keep parameters immutable

Why?

Mutations are side-effects and keep a function impure i.e every time you execute it, it not only returns what it is supposed to but changes state outside of its control i.e no two invocations will be the same. Additionally, it makes it worse if the mutated input needs to be reused as state by another unit of code since the state has now changed inexplicably. Avoid mutations as much as possible!

1 step:

1. **Step 1:** Copy things before mutating them

```
// Step 1: Copy things before mutating them
// -------
// Not mutating an input parameter
function getBillableInvoices(userInvoices, usersWithFreeTrial) {
  // Gets a list of all key-value pairs in the userInvoices map
  return userInvoices
    .entries()
    // filter() copies any values matching the condition into a new list
    .filter(entry => !usersWithFreeTrial.contains(entry.getKey()))
    .map(entry => entry.getValue())
} // Good
```

Handle critical inputs to functions

Why?

Doing nothing when a critical input is missing can cause surprises

1 step:

1. **Step 1:** Make critical inputs *required*!

```
// Step 1: Make critical inputs required
// -------
// Doing nothing when a critical input is missing can cause surprises
class UserDisplay {
  private messages: LocalizedMessages;
  // ...
  displayLegalDisclaimer(legalText: string?) {
    // legalText can be null.
    if (legalText == null) {
      return;
      // When legalText is null, the function returns without displaying anything.
    }

    displayOverlay(
      title: messages.getLegalDisclaimerTitle(),
      message: legalText,
      textColor: Color.RED
    );
  }
} // Bad


// Solution: make critical inputs required
class UserDisplay {
  private messages: LocalizedMessages;
  // ...
  displayLegalDisclaimer(legalText: string) {
    // legalText cannot be null.
    // A disclaimer will always be displayed
    displayOverlay(
      title: messages.getLegalDisclaimerTitle(),
      message: legalText,
      textColor: Color.RED
    );
  }
} // Good
// ...
function ensureLegalCompliance() {
  // Overhead check in a higher level abstraction but it is necessary:
  const signupDisclaimer = messages.getSignupDisclaimer();
  if (signupDisclaimer == null) {
    return false;
  }
  userDisplay.displayLegalDisclaimer(messages.getSignupDisclaimer());
}; // Good
```

Fix enum usage

Why?

Enums need to be future-proofed. Else, they can be problematic. If it is not robust enough to more values being added in the future, it will make development difficult.

1 step:

1. **Step 1:** Use an **exhaustive switch statement**
2. **Step 2:** Beware of the default case (including throwing an error from the default case)

```
// Step 1: Use an exhaustive switch statement
// -------
// Implicitly handling future enum values can be problematic
enum PredictedOutcome {
  COMPANY_WILL_GO_BUST,
  COMPANY_WILL_MAKE_A_PROFIT,
} // Two enum values
// ...
isOutcomeSafe(prediction: PredictedOutcome) {
  if (prediction === PredictedOutcome.COMPANY_WILL_GO_BUST) {
    return false; // COMPANY_WILL_GO_BUST explicitly handled as not safe
  }
  return true; // All other enum values implicitly handled as safe
} // Bad
// What if we need to add WORLD_WILL_END enum? A value indicating that the world is predicted to end.
// It is not safe to assume that any engineer adding an enum value to PredictedOutcome would be aware of the need to also update the isOutcomeSafe() function.
// And if  isOutcomeSafe() function is not updated, invoking it with WORLD_WILL_END could lead to a catastrophic outcome.


// Solution: Use an exhaustive switch statement
enum PredictedOutcome {
  COMPANY_WILL_GO_BUST,
  COMPANY_WILL_MAKE_A_PROFIT,
  // ... can add more values, such as WORLD_WILL_END
}
// ...
isOutcomeSafe(prediction: PredictedOutcome) {
  switch (prediction) { // Each enum value is explicitly handled.
    case COMPANY_WILL_GO_BUST:
      return false;
    case COMPANY_WILL_MAKE_A_PROFIT:
      return true;
  }
  // An unhandled enum value is a programming error, so an unchecked exception is
thrown.
  throw new UncheckedException("Unhandled prediction: " + prediction);
} // Good
```

```
// Step 2: Beware of the default case (including throwing an error from the default
case)
// -------
isOutcomeSafe(prediction: PredictedOutcome) {
  switch (prediction) { // Each enum value is explicitly handled.
    case COMPANY_WILL_GO_BUST:
      return false;
    case COMPANY_WILL_MAKE_A_PROFIT:
      return true;
    default:
      return false; // Better to throw an exception in the default case
  }
} // Neutral
```

Make code hard to misuse

Fix immutability within classes

Note: Some of these concepts might apply to functions as well!

2 steps:

1. **Step 1:** Set values only at construction time
    1. We can make a class immutable (and prevent it being misused) by ensuring that all the values are provided at construction time and that they cannot be changed after this.
2. **Step 2:** Use a design pattern for **immutability** (when you cannot always set values only at construction time)
    1. Use the **Builder pattern** (When some values that a class can be constructed with are optional, it can become quite unwieldy to specify them all in the constructor. Rather than making the class mutable by adding setter functions, it can often be better to use the builder pattern)
        1. A builder class that allows values to be set *one by one*
        2. An *immutable, read-only version* of the class that is built from the builder
    2. Use the **Copy-on-write pattern**

3. **Step 3:** Make things deeply immutable by **defensively copying things** (especially *lists*)
4. **Step 4:** Use **immutable data structures**

```
// Step 1: Set values only at construction time
// -------
// Mutable classes can be easy to misuse
class TextOptions {
  private font: Font;
  private fontSize: number;
  constructor(font: Font, fontSize: number) {
    this.font = font;
    this.fontSize = fontSize;
  }
  setFont(font: Font) { // The font can be changed at any time by calling setFont().
    this.font = font;
  }
  setFontSize(fontSize: number) { // The font size can be change at any time by
calling setFontSize()
    this.fontSize = fontSize;
  }
  getFont(): Font {
    return font;
  }
  getFontSize(): number {
    return fontSize;
  }
} // Bad
const fontArial14 = new TextOptions('Arial', 14);
fontArial14.setFont('Verdana');
fontArial14.setFontSize(10);
fontArial14.getFont(); // Verdana
fontArial14.getFontSize(); // 10
// Bad


// Mutable classes can be easy to misuse
class TextOptions {
  // If we can mark the properties as 'final' in a language, that would further
prevent misuse
  private font: Font;
  private fontSize: number;
  constructor(font: Font, fontSize: number) {
    this.font = font;
    this.fontSize = fontSize;
  }
  getFont(): Font {
    return font;
  }
  getFontSize(): number {
    return fontSize;
  }
} // Good
const fontArial14 = new TextOptions('Arial', 14);
fontArial14.getFont(); // Arial
fontArial14.getFontSize(); // 14
// Good
```

```
// Step 2: Use a design patterm for immutability
// -------
// (a) The Builder pattern
class TextOptions {
  private font: Font;
  private fontSize: number;
  constructor(font: Font, fontSize: number) {
    this.font = font;
    this.fontSize = fontSize;
  }
  getFont(): Font {
    return font;
  }
  getFontSize(): number {
    return fontSize;
  }
}
class TextOptionsBuilder {
  private font: Font;
  private fontSize: number;
  constructor(font: Font) {
    this.font = font;
  }
  setFontSize(fontSize: number): TextOptionsBuilder {
    this.fontSize = fontSize;
    return this;
  }
  build(): TextOptions {
    return new TextOptions(font, fontSize);
  }
} // Good
const myTextOptions = new TextOptionsBuilder(Font.ARIAL);
// Can set values later ...
myTextOptions
  .setFontSize(12.0)
  .build(); // ... but once you build, it cannot be mutated!
// Good



// (b) The Copy-on-Write pattern
// Useful when you have a language that overrides a constructor based on type and
number of arguments
// Not really applicable to JS/TS but useful to know that it is an alternative to the
builder pattern
```

```
// Step 3: Make things deeply immutable by defensively copying things (especially
lists)
// -------
class TextOptions {
  private fontFamily: Font[];
  private fontSize: number;
  constructor(fontFamily: Font[], fontSize: number) {
    // A copy of the fontFamily list that only this class has a reference to
    this.fontFamily = lodash.cloneDeep(fontFamily);
    this.fontSize = fontSize;
  }
  getFontFamily(): Font[] {
    // A copy of the fontFamily list that is generated for external code
    return lodash.cloneDeep(fontFamily);
  }
  getFontSize() {
    return fontSize;
  }
} // Good
```

```
// Step 4: Use immutable data structures
// -------
// To maintain non-primitives in an immutable way so that we interact with these data
structures via its APIs only - so as to preserve immutability
// JavaScript-based language—A couple of options are the following:
// – The List class from the Immutable.js module (http://mng.bz/pJAR)
// – A JavaScript array, but with the Immer module used to make it immutable
(https://immerjs.github.io/immer/)
```

Fix data types

5 steps:

1. **Step 1:** Use a **dedicated type** and avoid overly general types, especially "pair types" that are easy to misuse
2. **Step 2:** Use a dedicated **library** for handling **date and time** (they maintain the data types and methods)
   1. **Do not use integers** for date and time
   2. It is difficult to differentiate between an instance of time and duration of time. It is also difficult to keep units consistent and timezones in mind

```
// Step 1: Avoid overly general types
// -------
// (a) Geenral type
// Ex: Using a list of list of number might seem quick to implement a list of latitude
and logitude pairs. But it has a number of drawbacks that make the code easy to
misuse. The following incorrect values are all accepted!
// 1. location = [longitude, latitude];
// 2. location = [51.178889]; ... many more.
class LocationDisplay {
  private map: DrawableMap;
  // ...
  // ***Semi-complicated documentation is required to explain the input parameter:***
  /**
   * Marks the locations of all the provided coordinates
   * on the map.
   *
   * Accepts a list of lists, where the inner list should
   * contain exactly two values. The first value should
   * be the latitude of the location and the second value
   * the longitude (both in degrees).
   */
  markLocationsOnMap(locations: number[][]): void {
    for (const location in locations) {
      map.markLocation(location[0], location[1]);
    }
  }
} // Bad


// (b) Pair types are bad too!
// Using Pair<Double, Double> instead of List<Double> solves some of the problems: the
pair has to contain exactly two values, so it prevents callers from accidentally
providing too few or too many values. But it does not solve the other problems:
// 1. The type List<Pair<Double, Double>> still does very little to explain itself
// 2. It’s still easy for an engineer to get confused about which way around the
latitude and longitude should be.
class Pair<A, B> {
  private first: A;
  private second: B;
  constructor(first: A, second: B) {
    this.first = first;
    this.second = second;
  }
  getFirst(): A {
    return first;
  }
  getSecond(): B {
    return second;
  }
}

// Solution: Use a dedicated type
// It can seem like a lot of effort or overkill to define a new class (or struct) to
represent something, but it’s usually less effort than it might seem and will save
engineers a lot of head scratching and potential bugs further down the line.
/**
 * Represents a latitude and longitude in degrees.
 */
class LatLong {
  private latitude: number;
  private longitude: number;
  constructor(latitude: number, longitude: number) {
    this.latitude = latitude;
    this.longitude = longitude;
  }
  getLatitude() {
    return latitude;
  }
  Double getLongitude() {
    return longitude;
  }
} // Good
```

```
// Step 2: Use a dedicated library for date and time (they maintain the data types and
methods)
// -------
// Some popular date and time libraries in JS:
// 1. Moment.js
// 2. Date-fns
// 3. Luxon
// 4. Day.js
// 5. Date.js
```

Fix multiple sources of truth

2 steps:

1. **Step 1:** Use **primary data** as the *single source of truth*
    1. *Primary data*: Things that need to be supplied to the code. There is no way that the code could work this data out without being told it
    2. *Derived data*: Things that the code can calculate based on the primary data
2. **Step 2:** Have a *single source of truth* for **logic**
    1. A given piece of code will usually *solve a high-level problem by breaking it down into a series of subproblems*
    2. Solve a sub-problem *once* and allow multiple high-level problem solutions to use that single sub-problem solution i.e reused

```
// Step 1: Use primary data as the single source of truth
// -------
class UserAccount {
  private credit: number;
  private debit: number;
  private balance: number;
  constructor(credit: number, debit: number, balance: number) {
    this.credit = credit; // primary data
    this.debit = debit; // primary data
    this.balance = balance; // Balance is derived data
    // Maintaining it is unnecessary since credit & debit can be used to calculate
balance
  }
  getCredit() {
    return credit;
  }
  getDebit() {
    return debit;
  }
  getBalance() {
    return balance;
  }
} // Bad


class UserAccount {
  private credit: number;
  private debit: number;
  constructor(credit: number, debit: number, balance: number) {
    this.credit = credit; // primary data
    this.debit = debit; // primary data
  }
  getCredit() {
    return credit;
  }
  getDebit() {
    return debit;
  }
  getBalance() { // Calculating derived data
    return credit - debit;
  }
} // Good!
// Note: When deriving data is expensive, we can. try CACHING it on other operations,
etc.
// i.e Optimize it.
```

```
// Step 2: Have a single source of truth for logic
// -------
class DataLogger {
  private loggedValues: number[];
  // ... (a) Logic to serialize values
  saveValues(file: FileHandler): void {
    const serializedValues = loggedValues
      .map(value => value.toString(Radix.BASE_10))
      .join(",");
    file.write(serializedValues);
  }
}
class DataLoader {
  // ... (b) Logic to de-serialize values from a file
  loadValues(file: FileHandler): number[] {
    return file.readAsString()
      .split(",")
      .map(str => Number.parseInt(str, Radix.BASE_10));
  }
} // Bad
```

```
// Both the DataLogger and DataLoader classes independently contain logic that specify
the format. When the classes both contain the same logic everything works fine, but if
one is modified and the other is not, problems will arise.


// Solution:
// DataLogger and DataLoader classes are both solving one of the same subproblems (the
format for storing serialized integers).
// We could make the code more robust and less likely to be broken by having a single
source of truth for the format for storing serialized integers. We can achieve this by
making the serialization and deserialization of a list of integers be a single,
reusable layer of code.
class IntListFormat {
  private DELIMITER: string = ",";
  private Radix: RADIX = Radix.BASE_10; // The delimiter and radix are specified in
constants.
  serialize(values: number[]) {
    return values
      .map(value => value.toString(RADIX))
      .join(DELIMITER);
  }
  deserialize(serialized: string): number[] {
    return serialized
      .split(DELIMITER)
      .map(str => Int.parse(str, RADIX));
  }
}
class DataLogger {
  private loggedValues: number[];
  private intListFormat: IntListFormat; // IntListFormat class used to solve the
subproblem
  // ...
  saveValues(file: FileHandler) {
    file.write(intListFormat.serialize(loggedValues));
  }
}
class DataLoader {
  private intListFormat: IntListFormat; // IntListFormat class used to solve the
subproblem
  // ...
  loadValues(file: FileHandler): number[] {
    return intListFormat.deserialize(file.readAsString());
  }
} // Good
```

Make code modular

Fix hardcoded and tightly coupled code

Why?

A high level problem (Ex: route planner) when tightly coupled to a sub-problem (Ex: North America map) such that the high level solution cannot make use of the sub-solution to a similar problem (Ex: applying route planner for an Africa map) then it is an issue sicne we cannot simply reconfigure the sub-solution to address the new high level problem.

2 steps:

1. **Step 1:** Use **Dependency Injection** to remove *hardcoding* and *tight coupling* of code
    1. Use *factory functions *to make it *easy* to dependency inject
    2. Design code with Dependency Injection in mind
2. **Step 2:** Depend on **interfaces** where possible (TS example)
    1. Depending on *concrete implemntations* limits adaptability
    2. Interfaces make code considerably more *modular* and *adaptable*

```
// Step 1: Use Dependency Injection to remove hardcoding and tight coupling of code
// -------
class RoutePlanner {
  roadMap;
  constructor() {
    this.roadMap = new NorthAmericaRoadMap(); // Tightly coupled / hardcoded
  }
  planRoute(startPoint, endPoint) {
    // ...
    // Cannot use a route planner for let's say, Asia!
  }
} // Bad

// (a) Basic
class RoutePlanner {
  roadMap;
  constructor(roadmap) {
    this.roadMap = roadmap; // Dependency injected
  }
  planRoute(startPoint, endPoint) {
    // ...
  }
} // Good
const northAmericaRoutePlanner = new RoutePlanner(new NorthAmericaRoadMap()); // Good
const asiaRoutePlanner = new RoutePlanner(new AsiaRoadMap()); // Good

// (b) Factory functions
class RoutePlanner {
  roadMap;
  constructor(roadmap) {
    this.roadMap = roadmap; // Dependency injected
  }
  planRoute(startPoint, endPoint) {
    // ...
  }
} // Good
class RoutePlannerFactory {
  static RoutePlanner createNorthAmericaRoadMap() {
    return new RoutePlanner(new NorthAmericaRoadMap()); // Good
  }
  static RoutePlanner createAsiaRoadMap() {
    return new RoutePlanner(new AsiaRoadMap()); // Good
  }
  // ...
} // Good
const northAmericaRoadMap = RoutePlannerFactory.createNorthAmericaRoadMap(); // Good
const asiaRoadMap = RoutePlannerFactory.createAsiaRoadMap(); // Good
```

```
// Step 2: Depend on interfaces where possible (TS example)
// -------
interface RoadMap { // Sub-problem to be solved: Creating a road map
  getRoads: Road[];
  getJunctions: Junction[];
};
class NorthAmericaRoadMap implements RoadMap { // Adaptable: N.A road map based on
interface
  // ...
};
class AsiaRoadMap implements RoadMap { // Adaptable: Asia road map based on interface
  // ...
};
class RoutePlanner {
  private roadMap: NorthAmericaRoadMap;
  constructor(roadmap: NorthAmericaRoadMap) { // BAD! Depends directly on
NorthAmericaRoadMap
    this.roadMap = roadmap;
  }
  // ...
} // Bad
const northAmericaRoutePlanner = new RoutePlanner(new NorthAmericaRoadMap());
// What about "new RoutePlanner(new AsiaRoadMap());" ? Not possible!
// Can only use an instance of NorthAmericaRoadMap
// Bad



interface RoadMap { // Sub-problem to be solved: Creating a road map
  getRoads: Road[];
  getJunctions: Junction[];
};
class NorthAmericaRoadMap implements RoadMap { // Adaptable: N.A road map based on
interface
  // ...
};
class AsiaRoadMap implements RoadMap { // Adaptable: Asia road map based on interface
  // ...
};
class RoutePlanner {
  private roadMap: RoadMap;
  constructor(roadmap: RoadMap) { // GOOD! Depends on a RoadMap interface
```

```
    // Now, any roadmap can be D.I'ed
    this.roadMap = roadmap;
  }
  // ...
} // Good
const northAmericaRoutePlanner = new RoutePlanner(new NorthAmericaRoadMap());
const asiaRoutePlanner = new RoutePlanner(new AsiaRoadMap());
// Good
```

Avoid class inheritance

Why?

Class inheritance can be problematic. It can:

- *Prevent clean layers of abstraction*: When one class inherits the other, it *inherits all the functionality of the super class* which means we might expose more functionality than we need to
- Make code hard to adapt

2 steps:

1. **Step 1:** Use **composition** over inheritance (Composition can be achieved by **dependency injection**)
    1. Cleaner layers of abstraction
    2. More adaptable code
2. **Step 2:** Handle **genuine *is-a* relationships** by using a *combination of interfaces and composition*
    1. Inheritance makes sense for pure *is-a* relationships
        1. ***But what happens if an is-a relationship exists for more than one superclass?*** (multiple inheritance)
        2. Ex: A *FlyingCar* can also be a *Car* and an *Airplane* at the same time
        3. There can be more such complicated relationships (fragile base class, diamond problem, etc) which make it tough to justify inheritance for pure *is-a* relationships.
    2. Interfaces define a hierarchy of objects and composition achieves reuse of code
        1. Use interfaces + composition to handle is-a relationships in case there is a chance of multiple inheritance
    3. Avoid *mixins* and *traits*

```
// Step 1: Use composition over inheritance
// -------
/* Utility for reading and writing from/to a file containing comma-separated values */
class CsvFileHandler implements FileValueReader, FileValueWriter {
  constructor(file: File) { /* ... */ }
  override getNextValue(): string? { /* ... */ }
  writeValue(value: string) { /* ... */ }
  close() { /* ... */ }
};
/* Utility for reading integers from a file one by one.
 * The file should contain comma-separated values. */
class IntFileReader extends CsvFileHandler { // INHERITANCE
```

```
  constructor(file: File) {
    super(file); // The IntFileReader constructor calls the superclass constructor.
  }
  getNextInt(): number? {
    const nextValue: string? = this.getNextValue();
    if (nextValue == null) {
      return null;
    }
    return Number.parseInt(nextValue, Radix.BASE_10);
  }
};
// Even though the IntFileReader extends CsvFileHandler
// and exposes the getNextInt() method, it also exposes other
// methods that are unnecessary! (Ex: Methods to write values)
// API of IntFileReader:
IntFileReader {
  getNextInt(): number? { /* ... */ } // Needed
  getNextValue(): string? { /* ... */  } // Not needed
  writeValue(String value) { /* ... */  } // Not needed
  close() { /* ... */  }
}; // Bad


// Inheritance makes it hard to adapt!
class SemicolonFileHandler implements FileValueReader, FileValueWriter {
  constructor(file: File) { /* ... */ }
  getNextValue(): string? { /* ... */ }
  writeValue(value: string) { /* ... */ }
  close() { /* ... */ }
}; // It implements the same FileValueReader, FileValueWriter interfaces as
CsvFileHandler
// Now, if IntFileReader wants to read integers from a file that uses semicolons
instead
// of commas, it is not as trivial to make the change as it sounds (due to
inheritance).
// Why?
// This is because IntFileReader inherits CsvFileHandler
// Since SemicolonFileHandler also contains the SAME METHODS as CsvFileHandler
// This would break existing solution of IntFileReader
// Workaround: Create a duplicate "SemicolonIntFileReader" class (Code duplication
(X))
// Bad!


// Composition:
class IntFileReader { // No inheritance
  private valueReader: FileValueReader; // Holds an instance to a FileValueReader
(COMPOSITION)
  constructor(valueReader: FileValueReader) { // Using D.I to enable COMPOSITION
    this.valueReader = valueReader;
  }
  getNextInt(): number? {
    cpnst nextValue: string? = valueReader.getNextValue();
    if (nextValue == null) {
      return null;
    }
    return Number.parseInt(nextValue, Radix.BASE_10);
  };
  close() {
    this.valueReader.close();
  }
};
// Possible since CsvFileHandler implements the FileValueReader interface:
const intFileReader = new IntFileReader(new CsvFileHandler());
const intFileReader = new IntFileReader(new SemicolonFileHandler()); // (1) Adaptable!
// Interface of IntFileReader exposes only what it needs ((2) Cleaner layer of
abstraction):
IntFileReader {
  getNextInt(): number?;
  close(): void;
} // It does not expose any getValue() or writeValue() methods that inheritance did!
// Good!
```

```
// Step 2: Handle genuine "is-a" relationships by using a combo of interfaces and
composition
// -------
class Car { /* .. */ };
class Airplane { /* ... */ };

class Volvo { /* ... */ }; // Justified: Volvo "is a" Car
class Boeing777 { /* ... */ }; // Justified: Boeing777 "is an" Airplane

class NewFlyingCar extends Car, Airplane {/* ... */ }; // Not justified: NewFlyingCar
"is an" Airplane as well as a Car - not a pure "is-a" relationship
// Bad


// Interface + Composition
interface Car {
  drive(): void;
};
interface Airplane {
  fly(): void;
}
class Volvo implements Car { /* ... */ };
class Boeing777 implements Airplane { /* ... */ };
class NewFlyingCar implements Car, Airplane {
  private drivingAction: Car;
  private flyingAction: Airplane;
  constructor(drivingAction, flyingAction) {
    this.drivingAction = drivingAction; // D.I to realise composition
    this.flyingAction = flyingAction; // D.I to realise composition
  };
  // ...
};
const myFlyingCar = new NewFlyingCar(new Volvo(), new Boeing777());
// Good!
```

Make classes care about themselves

Why?

Single Responsibility Principle (SRP): A class (or unit of code such a function) should be responsible for only one concept. It should not reach for details present in higher or lower levels of abstraction. Ex: worrying about implementation details for a sub-problem; It should be delegated to a new unit of code (class / function) that appears at a lower level of abstraction.

If we do not follow SRP, a single concept gets spread across multiple classes and require modifications in multiple places related to that concept with might result in more bugs if not handled comprehensively. Or, a single class might bloat in logic.

2 steps:

1. **Step 1:** Make a class not worry about **implementation details** for a **sub-problem**
   1. Use the **Law of Demeter (LoD)** to spot if a class is interacting with objects other than it's immediately related to
   2. *If a class cares too much about another class, it is a code smell!*
2. **Step 2: Group related data** into objects or classes (**Encapsulation**)
   1. When different pieces of data are *inescapably related to one another*, and there’s no practical scenario in which someone would want some of the pieces of data without wanting all of them, then it usually makes sense to encapsulate them together.
   2. Higher levels of abstraction i.e functions or classes making use of this data as a whole need not have specific knowledge of individual pieces of this data to pass the grouped data around.

```
// Step 1: Make a class not worry about implementation details for a sub-problem
// -------
class Book {
  chapters;
  wordCount() {
    return chapters.map(getChapterWordCount).sum();
  }
  static getChapterWordCount(chapter) {
    // Code smell: Book cares too much about a Chapter
    // (OR) Book goes into implementation details of Chapter
    // Ex: (chapter -> get preludes (ok) -> wordcount (bad))
    // Ex: (chapter -> sections (ok) -> word count (bad))
    return chapter.getPrelude().wordCount() +
        chapter.getSections().map(section -> section.wordCount()).sum();
  }
}
class Chapter {
  getPrelude() { /* ... */ }
  getSections() { /* ... */ }
} // Bad
```

```
class Book {
  chapters;
  wordCount() {
    // Book cares only about Chapter and not lower details
    // It only uses the methods exposed by what Chapter wants to expose. (1st level)
    // Ex: (chapter -> wordcount (ok))
    return chapters
      .map(chapter -> chapter.wordCount())
      .sum();
  }
}
class Chapter {
  // ...
  getPrelude() { /* ... */ }
  getSections() { /* ... */ }
  wordCount() { /* ... */ }
} // Good

/* Law of Demeter (alternate way to detect this code smell) */
// *** An object should interact only with other objects that it’s immediately related
to!***
// A book (level 2) caring about a chapter's (level 1) method such as wordCount() is
an immediate interaction
// A book (level 3) caring about a chapter's (level 2) sections and each section's
(level 1) word count is not an immediate interaction
```

```
// Step 2: Group related data into objects or classes (Encapsulation)
// -------
class TextBox {
  // ...
  renderText(
      text,
      font,
      fontSize,
      lineHeight,
      textColor
  ) {
    // ...
  }
}
class UiSettings {
  // ...
  getFont() { /* ... */ }
  getFontSize() { /* ... */ }
  getLineHeight() { /* ... */ }
  getTextColor() { /* ... */ }
}
class UserInterface {
  messageBox; // An instance of TextBox
  uiSettings; // An instance of UiSettings
  // ...
  displayMessage(String message) {
    // Needs to have specific knowledge of text styling! (X)
    messageBox.renderText(
      message,
      uiSettings.getFont(),
      uiSettings.getFontSize(),
      uiSettings.getLineHeight(),
      uiSettings.getTextColor()
    );
  }
} // Bad



// New class to "encapsulate" / "group" related data:
class TextOptions { // Encapsulate related data + methods to read/write/interact with
them!
  font;
  fontSize;
  lineHeight;
  textColor;
  TextOptions(font, fontSize, lineHeight, textColor) {
    this.font = font;
    this.fontSize = fontSize;
    this.lineHeight = lineHeight;
    this.textColor = textColor;
  }
  getFont() { return font; }
  getFontSize() { return fontSize; }
  getLineHeight() { return lineHeight; }
  getTextColor() { return textColor; }
}
class TextBox {
  // ...
  renderText(text, textStyle) {
    // ... takes care of picking specific styles from textStyleto use for styling
  }
};
class UiSettings {
  // ...
  textOptions;
  constructor(textOptions) { // Accepts an instance of textOptions (D.I)
    this.textOptions = textOptions;
  }
  getTextStyle() { /* ... */ }
}
class UserInterface {
  messageBox; // An instance of TextBox
  uiSettings; // An instance of UiSettings
  displayMessage(message) { // Has no specific knowledge of text styling
    messageBox.renderText(message, uiSettings.getTextStyle()); // Better than setting
each text style
    // Just pass the test styles together.
  }
}; // Good
```

Avoid leaking implementation details

Why?

- If implementation details are leaked, this can reveal things about lower layers in the code and can make it *very hard to modify or reconfigure things in the future*.
- One of the most common ways code can leak an implementation detail is by returning a type that is tightly coupled to that detail.

2 steps:

1. **Step 1:** Return a type appropriate to the layer of abstraction (TS example)
    1. It’s very hard to ever change the implementation of the functioning returning a vsalue based on an implementation detail
    2. Higher levels of abstraction, i.e functions or classes using it, need to account for the implementation detail (tedious, tightly coupled, and unnecessary)
2. **Step 2:** Make exceptions appropriate to the layer of abstraction (TS example)
    1. In order to prevent implementation details being leaked, each layer in the code should ideally reveal only error types that reflect the given layer of abstraction.
    2. We can achieve this by wrapping any errors from lower layers into error types appropriate to the current layer. This means that callers are presented with an appropriate layer of abstraction while also ensuring that the original error information is not lost (because it’s still present within the wrapped error).

```
// Step 1: Return a type appropriate to the layer of abstraction
// -------
// The "ProfilePictureService" is implemented using an "HttpFetcher" to fetch the
profile picture data from a server. The fact that "HttpFetcher" is used is an
**implementation detail**, and as such, any engineer using the "ProfilePictureService"
class should ideally NOT have to concern themselves with this piece of information.
class ProfilePictureService {
  private httpFetcher: HttpFetcher;
  getProfilePicture(userId: Int64): ProfilePictureResult { /* ... */ }
}
class ProfilePictureResult {
  // ...
  /**
   * Indicates if the request for the profile picture was successful or not.
   */
  getStatus(): HttpResponse.Status { /* ... */ }
  /**
   * The image data for the profile picture if it was successfully found.
   */
  getImageData(): HttpResponse.Payload? { /* ... */ }
} // Bad


class ProfilePictureService {
  private httpFetcher: HttpFetcher;
  getProfilePicture(userId: Int64): ProfilePictureResult { /* ... */ }
}
// A custom enum to define just the statuses we require:
// (Good -> appropriate to level of abstraction)
enum Status {
    SUCCESS,
    USER_DOES_NOT_EXIST,
    OTHER_ERROR,
}
class ProfilePictureResult {
  // ...
  /**
   * Indicates if the request for the profile picture was successful or not.
   */
  getStatus(): Status { /* ... map an http status to one of the enums we require ...
*/ }
  /**
   * The image data for the profile picture if it was successfully found.
   */
  // Sends an image (bytes) instead of a custom payload type:
  getImageData(): Byte { /* ... Send an image back instead of payload from service ...
*/ }
} // Good
```

```
// Step 2: Make exceptions appropriate to the layer of abstraction
// -------
// The lower layer is the "TextImportanceScorer" interface, and the upper layer is the
"TextSummarizer" class. In this scenario, "ModelBasedScorer" is a concrete
implementation that implements the "TextImportanceScorer" interface, but
"ModelBasedScorer.isImportant()" can throw the unchecked exception
"PredictionModelException".
class TextSummarizer {
  private importanceScorer: TextImportanceScorer;
  // ...
  summarizeText(text: string): string {
    return paragraphFinder
      .find(text)
      .filter(paragraph => importanceScorer.isImportant(paragraph))
      .join("\n\n");
  }
};
interface TextImportanceScorer {
  isImportant(text: string): boolean;
};
class ModelBasedScorer: TextImportanceScorer {
  // ...
  /**
   * @throws PredictionModelException if there is an error
   * running the prediction model.
   */
  isImportant(text: String): boolean {
    return model.predict(text) >= MODEL_THRESHOLD;
  }
} // Bad!

// ...
updateTextSummary(ui: UserInterface) {
  userText = ui.getUserText();
  try {
    summary = textSummarizer.summarizeText(userText);
    ui.getSummaryField().setValue(summary);
  } catch (e: PredictionModelException) { // PredictionModel- Exception caught and
handle
    ui.getSummaryField().setError("Unable to summarize text");
  }
};
// ...
// Bad!



// New error type created for the current level of abstraction
// Wraps the errors it receives:
class TextSummarizerException extends Exception {
  // ...
  TextSummarizerException(cause: Throwable) { /* ... */ }
};
class TextSummarizer {
  private importanceScorer: TextImportanceScorer;
  // ...
  summarizeText(text: string) {
    try {
      return paragraphFinder
        .find(text)
        .filter(paragraph => importanceScorer.isImportant(paragraph))
        .join("\n\n");
    } catch (e: TextImportanceScorerException) {
      throw new TextSummarizerException(e); // Throwing an error that is not an
implementation detail!
    }
  }
}
// New error type created for the current level of abstraction
class TextImportanceScorerException extends Exception {
  // ...
  TextImportanceScorerException(cause: Throwable) { /* ... */ }
}
```

```
interface TextImportanceScorer {
  isImportant(String text): boolean;
}
class ModelBasedScorer implements TextImportanceScorer {
  // ...
  isImportant(text: string): boolean {
    try {
      return model.predict(text) >= MODEL_THRESHOLD;
    } catch (PredictionModelException e) {
      throw new TextImportanceScorerException(e); // Throwing an error that is not an
implementation detail!
    }
  }
} // Good
```

Make code reusable and generalizable

Identify assumptions in code and fix it

2 steps:

1. **Step 1:** Avoid unnecessary assumptions
2. **Step 2:** If an assumption is necessary, **enforce** it!
    1. Alternatively, you can use an **error signaling technique**

```
// Step 1: Avoid unnecessary assumptions
// -------
class Article {
  private sections = [];
  // ...
  getAllImages() {
    for (section in sections) {
      if (section.containsImages()) {
        // There should only ever be a maximum of one
        // section within an article that contains images. <-- [An unnecessary
assumption]
        return section.getImages();
      }
    }
    return [];
  }
} // Bad

class Article {
  private sections = [];
  // ...
  getAllImages() {
    let images = [];
    for (section in sections) {
      if (section.containsImages()) {
        images.push(...section.getImages());
        // Might take more time to loop through all sections.
        // However, it is better than making an unnecessary assumption.
      }
    }
    return [];
  }
} // Good
```

```
// Step 2: If an assumption is necessary, enforce it!
// -------
class Article {
  private sections = [];
  // ...
  getImageSection() {
    // There should only ever be a maximum of one
    // section within an article that contains images.
    return sections
        .filter(section => section.containsImages())[0];
  }
}
class ArticleRenderer {
  // ...
  render(article) {
    // ...
    imageSection = article.getImageSection(); // Can handle only one image containing
section!!
    if (imageSection != null) {
      templateData.setImageSection(imageSection);
    }
  }
} // Bad


class Article {
  private sections = [];
  // ...
  getOnlyImageSection() { // Fn. name conveys the assumption callers are making
    const imageSections = sections
        .filter(section => section.containsImages());

    // An assertion enforces the assumption
    assert(imageSections.length <= 1, "Article contains multiple image sections") //
Utility method

    return imageSections[0];
  }
} // Good
```

Fix global state

Why?

Global state is bad because it affects every context within a program and using them makes the implicit assumption that no one would ever want to reuse the code for a slightly different purpose, which is wrong!

- Makes variable reuse unsafe

Identifying global state:

1. **Static** variables (static properties and methods of classes)
2. **File-level variables**
3. Defining **properties on the global object** (Ex: window)

1 step:

1. **Step 1: Dependency-inject** shared state
    1. Convert a static class into one that needs to be instantiated
    2. Dependency inject this instance, i.e state, where it needs to be used

```
// Step 1: Dependency-inject shared state
// -------
class ShoppingBasket {
  static items = [];
  static addItem(item) {
    items.add(item);
  }
  static getItems() {
    return [...items]; // copy of items
  }
}
class ViewItemWidget {
  // ...
  addItemToBasket(item) {
    ShoppingBasket.addItem(item); // Modifies global state
  }
  // ...
  viewBasketWidget() {
    items = ShoppingBasket.getItems(); // Reads global state
    // ...
  }
} // Bad
// If someone else tries to access the basket,
// 1. Inconsistent state might be the result (Another user might have removed / added
items)
// 2. There is no segregation of users' baskets



class ShoppingBasket {
  items = []; // No more static
  addItem(item) { // No more static
    items.add(item);
  }
  getItems() { // No more static
    return [...items]; // copy of items
  }
}
```

```
class ViewItemWidget {
  item;
  basket;
  constructor(item, basket) { // Dependency inject shared state (basket)
    this.item = item;
    this.basket = basket;
  }
  // ...
  addItemToBasket(item) {
    basket.addItem(item);
  }
  // ...
  viewBasketWidget() {
    items = ShoppingBasket.getItems();
    // ...
  }
} // Good
const normalBasket = new ShoppingBasket();
const normalWidgetItem = new ViewItemWidget("item 1", normalBasket); // D.I
const fruitBasket = new ShoppingBasket();
const fruitWidgetItem = new ViewItemWidget("apple", fruitBasket); // D.I
// Good
```

Fix reusability in low-level code

2 steps:

1. **Step 1:** Provide sensible defaults *only* in **high-level code** *(i.e assumptions are more likely to hold here)*
    1. *Do not provide assumptions in low-level code*. They are usually unnecessary and inflexible for reuse.
2. **Step 2:** Make functions take only what they need as parameters
3. **Step 3:** Use **generics** to allow for generalizability that a specific type limits (TS example)
    1. It would be a lot better if the code generalized to solve almost identical subproblems

```
// Step 1: Provide sensible defaults only in high-level code (assumptions are likely
to hold)
// -------
class UserDocumentSettings {
  private font;
  getPreferredFont() {
    if (font != null) {
      return font;
    }
    // Default of Font.ARIAL
    return Font.ARIAL;
  }
} // Bad
// 1. UserDocumentSettings is low-level code (in this example)
// 2. If another high-level piece of code uses userDocumentSettings.getPreferredFont()
```

```
and receives a default value of ARIAL, there is no way for it to change that!!
// 3. Since UserDocumentSettings is low-level abstraction, there is a high chance that
a higher-level of abstraction piece of code will want to use it and with a different
default / without one.

class UserDocumentSettings { // low level of abstraction
  private font;
  getPreferredFont() {
    return font || null; // Returns null if there is no font i.e no default
  }
}
class DefaultDocumentSettings { // high level of abstraction [Helper for defaults]
  // ...
  getFont() {
    return Font.ARIAL;
  }
}
class DocumentSettings {  // high level of abstraction
  // Default settings for the whole document is high level abstraction
  // Highly unlikely that someone with use this class for a different purpose!
  getFont() {
    userFont = userDocumentSettings.getPreferredFont();
    // Overhead, but worth it:
    if (userFont != null) {
      return userFont;
    }
    return defaultDocumentSettings.getFont(); // Using default in high level
abstraction
  }
} // Good
```

```
// Step 2: Make functions take only what they need as parameters
// -------
class TextBox {
  textContainer;
  //...
  setTextStyle(options) {
    setFont(...);
    setFontSize(...);
    setLineHight(...);
    setTextColor(options);
  }
  setTextColor(options) { // Bad, we only need a text color to set!
    textContainer.setStyleProperty("color", options.getTextColor().asHexRgb());
  }
} // Bad

class TextBox {
  textElement;
  // ...
  setTextStyle(options) {
    setFont(...);
    setFontSize(...);
    setLineHight(...);
    setTextColor(options.getTextColor()); // Good!
  }
  setTextColor(color) {
    textElement.setStyleProperty("color", color.asHexRgb());
  }
}
// ...
styleAsWarning(textBox) {
  textBox.setTextColor(Color.RED);
} // Good
```

```
// Step 3: Use generics to allow for generalizability that a specific type limits (TS
example)
// -------
class RandomizedQueue {
  private values: string[]; // BAD! TIGHTLY COUPLED TO A QUEUE OF STRINGS!
  add(string value) {
    values.add(value);
  }
  /**
   * Removes a random item from the queue and returns it.
   */
  getNext(): string? {
    if (values.isEmpty()) {
      return null;
    }
    const randomIndex = generateRandomInt(0, values.size());
    values.swap(randomIndex, values.size() - 1);
    return values.removeLast();
  }
} // Bad
// Cannot use it for maintaining a RandomizedQueue of numbers,images, etc.
// i.e Similar sub-problem but on a slightly different data type.

class RandomizedQueue<T> {
  private values: <T>[]; // GOOD! NOT COUPLED TO A QUEUE OF ONLY STRINGS!
  add(string value) {
    values.add(value);
  }
  /**
   * Removes a random item from the queue and returns it.
   */
  getNext(): T? {
    if (values.isEmpty()) {
      return null;
    }

    const randomIndex = generateRandomInt(0, values.size());
    values.swap(randomIndex, values.size() - 1);
    return values.removeLast();
  }
} // Good
// Can use it for maintaining a RandomizedQueue of numbers, images, etc.
```