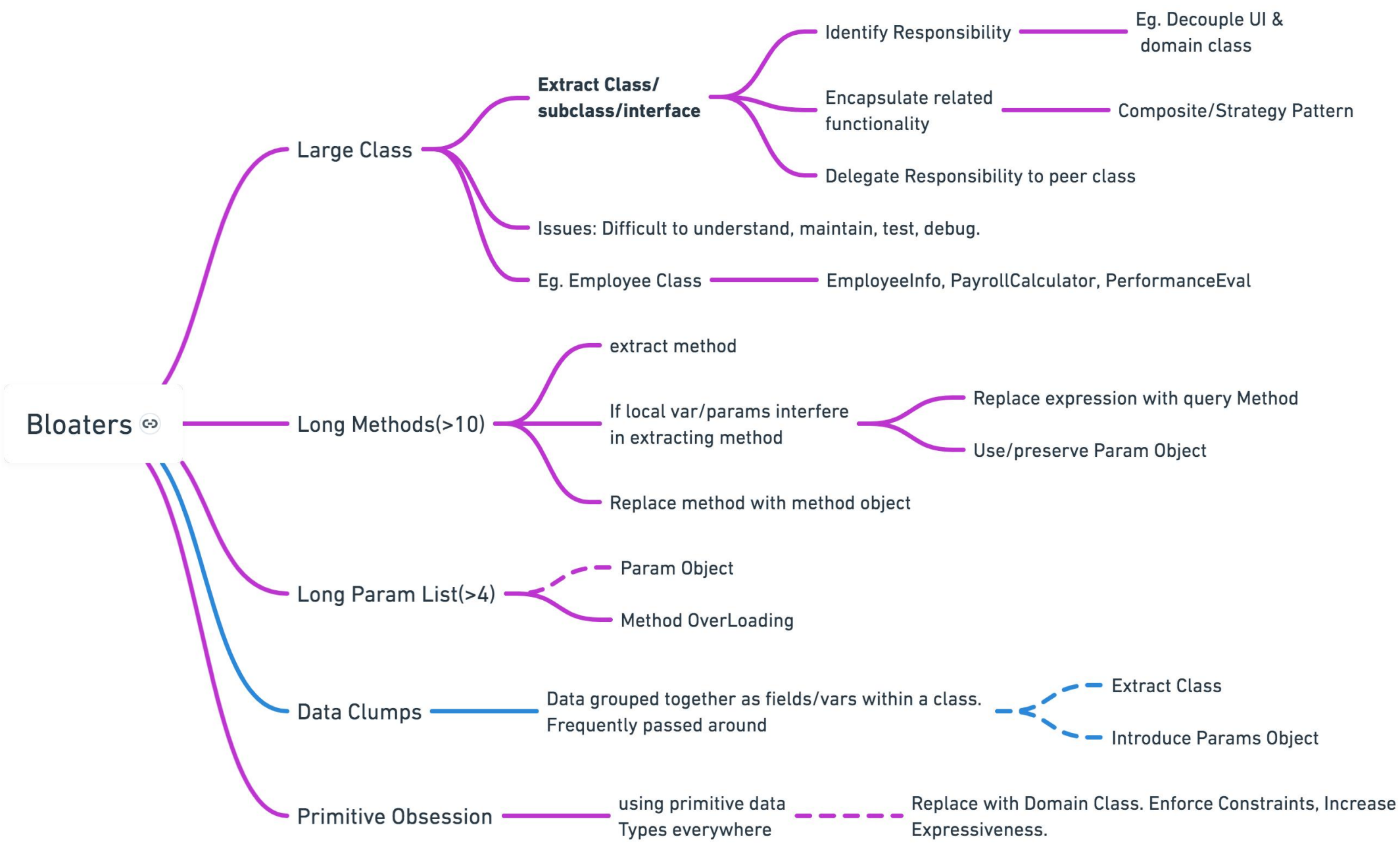

Bloaters
Large Class
Extract Class/ subclass/interface
Identify Responsibility
Eg. Decouple UI & domain class
Encapsulate related functionality
Composite/Strategy Pattern
Delegate Responsibility to peer class
Issues: Difficult to understand, maintain, test, debug.
Eg. Employee Class
EmployeeInfo, PayrollCalculator, PerformanceEval
Long Methods(>10)
extract method
If local var/params interfere in extracting method
Replace expression with query Method
Use/preserve Param Object
Replace method with method object
Long Param List(>4)
Param Object
Method OverLoading
Data Clumps
Data grouped together as fields/vars within a class. Frequently passed around
Extract Class
Introduce Params Object
Primitive Obsession
using primitive data Types everywhere
Replace with Domain Class. Enforce Constraints, Increase Expressiveness.

```java
// Original Large Class
public class Employee {
    private String name;
    private int age;
    private double salary;

    // Other fields and methods related to employee information

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public void calculatePayroll() {
        // Payroll calculation logic
    }

    public void evaluatePerformance() {
        // Performance evaluation logic
    }

    // Other methods and fields related to employee management
}
```

```java
// EmployeeInformation class focusing on basic information
public class EmployeeInformation {
    private String name;
    private int age;
    private double salary;

    public EmployeeInformation(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    // Getters and setters for basic information
}

// PayrollCalculator class focusing on payroll calculations
public class PayrollCalculator {
    public void calculatePayroll(EmployeeInformation employee) {
        // Payroll calculation logic
    }
}

// PerformanceEvaluator class focusing on performance evaluations
public class PerformanceEvaluator {
    public void evaluatePerformance(EmployeeInformation employee) {
        // Performance evaluation logic
    }
}
```

Large Class

```java
public class OrderProcessor {
    private String customerName;
    private String shippingAddress;
    private String billingAddress;
    // Other fields related to orders

    public void processOrder() {
        // Processing logic using customer details
        System.out.println("Processing order for " + customerName);
        System.out.println("Shipping to: " + shippingAddress);
        System.out.println("Billing to: " + billingAddress);
        // Additional order processing logic
    }

    // Other methods and fields related to orders
}
```

```java
public class OrderProcessor {
    private CustomerInfo customerInfo;
    // Other fields related to orders

    public void processOrder() {
        // Processing logic using encapsulated customer details
        System.out.println("Processing order for " + customerInfo.get
        System.out.println("Shipping to: " + customerInfo.getShipping
        System.out.println("Billing to: " + customerInfo.getBillingAd
        // Additional order processing logic
    }

    // Other methods and fields related to orders
}

public class CustomerInfo {
    private String customerName;
    private String shippingAddress;
    private String billingAddress;
    // Additional customer-related fields

    public CustomerInfo(String customerName, String shippingAddress,
        this.customerName = customerName;
        this.shippingAddress = shippingAddress;
        this.billingAddress = billingAddress;
        // Additional initialization logic
    }

    // Getters for customer details

    // Additional methods related to customer info, if needed
}
```

Data Clumps

```java
public double calculateRectangleArea(double length, double width) {
    // Calculation logic
    return length * width;
}
```

efactoring this to use a dedicated `Rectangle` class:

```java
public class Rectangle {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double calculateArea() {
        return length * width;
    }
}
```

Primitive Obsessions

```java
public double calculateTotalPrice(double basePrice, double taxRate,
                                  double discountPercentage, boolean
                                  boolean isTaxExempt) {
    // Calculation logic
    // ...
}
```

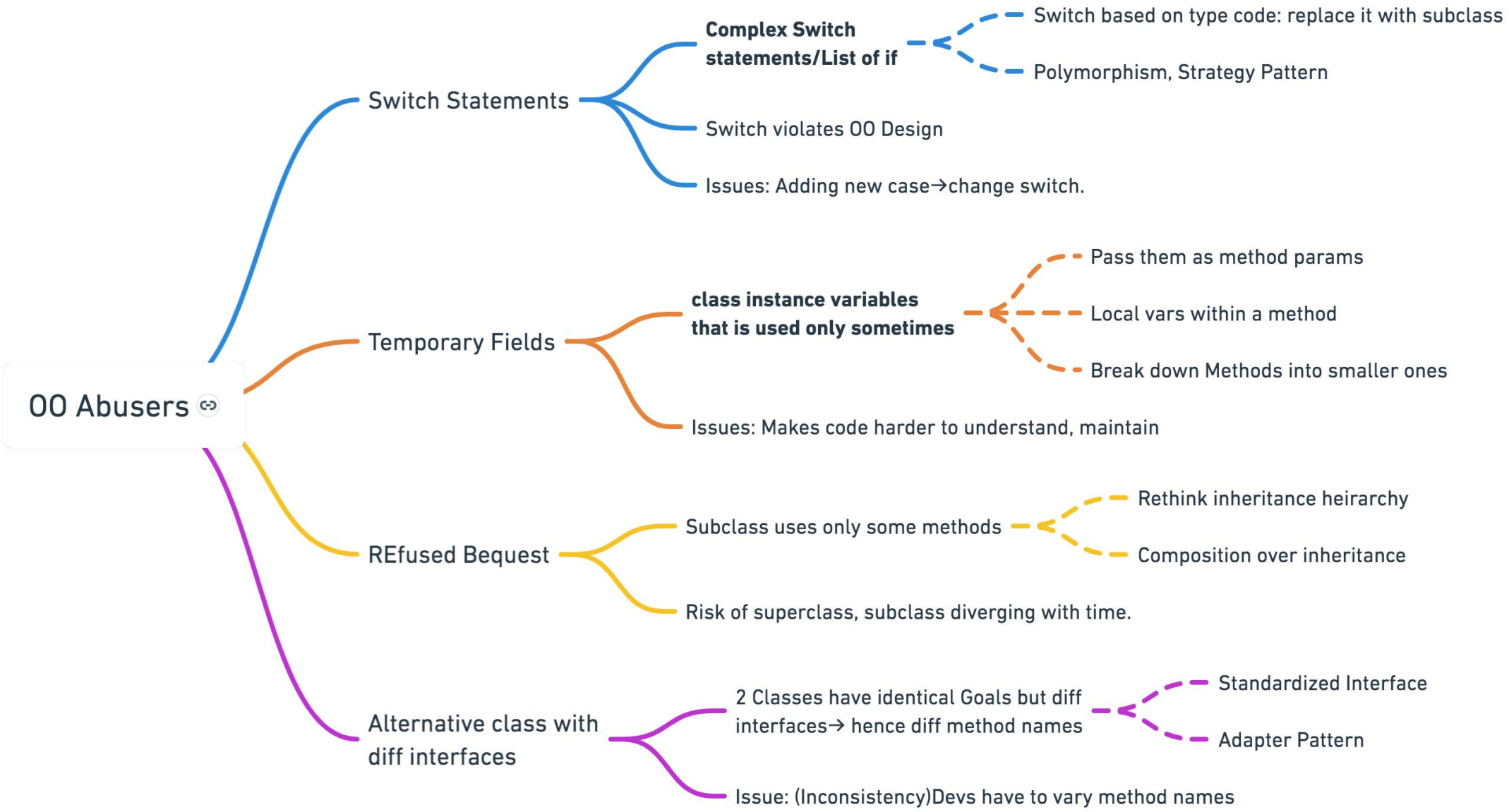
Let's extract the parameter list:

```java
public class OrderCalculationParameters {
    private double basePrice;
    private double taxRate;
    private double discountPercentage;
    private boolean isDiscounted;
    private boolean isTaxExempt;

    public OrderCalculationParameters(double basePrice, double taxRat
        this.basePrice = basePrice;
        this.taxRate = taxRate;
        this.discountPercentage = discountPercentage;
        this.isDiscounted = isDiscounted;
        this.isTaxExempt = isTaxExempt;
    }

    // Getters for individual parameters (optional, depending on usag
}
```

Large Params List

```java
public class OrderProcessor {

    public void processOrder(Order order) {
        // Step 1: Validate order
        if (!isValidOrder(order)) {
            logError("Invalid order");
            return;
        }

        // Step 2: Calculate total price
        double totalPrice = calculateTotalPrice(order);

        // Step 3: Apply discounts
        applyDiscounts(order, totalPrice);

        // Step 4: Update inventory
        updateInventory(order);

        // Step 5: Notify user
        sendOrderConfirmation(order);

        // ... additional steps

        // Step 6: Log order details
        logOrderDetails(order, totalPrice);
    }

    // ... other methods for validation, calculation, discount applic
}
```

Large Method


OO Abusers
Switch Statements
Complex Switch statements/List of if
Switch based on type code: replace it with subclass
Polymorphism, Strategy Pattern
Switch violates OO Design
Issues: Adding new case→change switch.
Temporary Fields
class instance variables that is used only sometimes
Pass them as method params
Local vars within a method
Break down Methods into smaller ones
Issues: Makes code harder to understand, maintain
REfused Bequest
Subclass uses only some methods
Rethink inheritance heirarchy
Composition over inheritance
Risk of superclass, subclass diverging with time.
Alternative class with diff interfaces
2 Classes have identical Goals but diff interfaces→ hence diff method names
Standardized Interface
Adapter Pattern
Issue: (Inconsistency)Devs have to vary method names

Switch statements

```
// Before
switch (type) {
    case "A":
        // logic for A
        break;
    case "B":
        // logic for B
        break;
    // ...

// After
interface Handler {
    void handle();
}

class AHandler implements Handler {
    @Override
    public void handle() {
        // logic for A
    }
}

class BHandler implements Handler {
    @Override
    public void handle() {
        // logic for B
    }
}
```

```
// Before
switch (strategyType) {
    case "A":
        // logic for A
        break;
    case "B":
        // logic for B
        break;
    // ...

// After
interface Strategy {
    void execute();
}

class AStrategy implements Strategy {
    @Override
    public void execute() {
        // logic for A
    }
}

class BStrategy implements Strategy {
    @Override
    public void execute() {
        // logic for B
    }
}
```

Refused Bequest

```
// Before
public class Animal {
    public void eat() {
        // logic for eating
    }

    public void sleep() {
        // logic for sleeping
    }
}

public class Dog extends Animal {
    @Override
    public void eat() {
        // logic for eating, specific to dogs
    }
}

// After
public class Dog {
    public void eat() {
        // logic for eating, specific to dogs
    }
}
```

```
// Using Inheritance

class Window {
    void draw() {
        // logic for drawing a window
    }

    void resize() {
        // logic for resizing a window
    }
}

class SpecializedWindow extends Window {
    void drawSpecialContent() {
        // logic for drawing specialized content
    }
}
```

```
// Using Composition

class Window {
    void draw() {
        // logic for drawing a window
    }

    void resize() {
        // logic for resizing a window
    }
}

class SpecializedContent {
    void drawSpecialContent() {
        // logic for drawing specialized content
    }
}

class SpecializedWindow {
    private Window window = new Window();
    private SpecializedContent specializedContent = new SpecializedC

    void draw() {
        window.draw();
        specializedContent.drawSpecialContent();
    }

    void resize() {
        window.resize();
    }
}
```

Temp Field

```
// Before
public class MyService {
    private int temporaryField;

    public void process() {
        temporaryField = 42;
        // logic using temporaryField
    }
}

// After
public class MyService {
    public void process() {
        int temporaryVariable = 42;
        // logic using temporaryVariable
    }
}
```

```
// Before
public void complexOperation() {
    // logic using temporaryField
    // ...
}

// After
public void complexOperation() {
    doSomething();
    // ...
}

private void doSomething() {
    int temporaryVariable = 42;
    // logic using temporaryVariable
}
```

Conditional Complexity

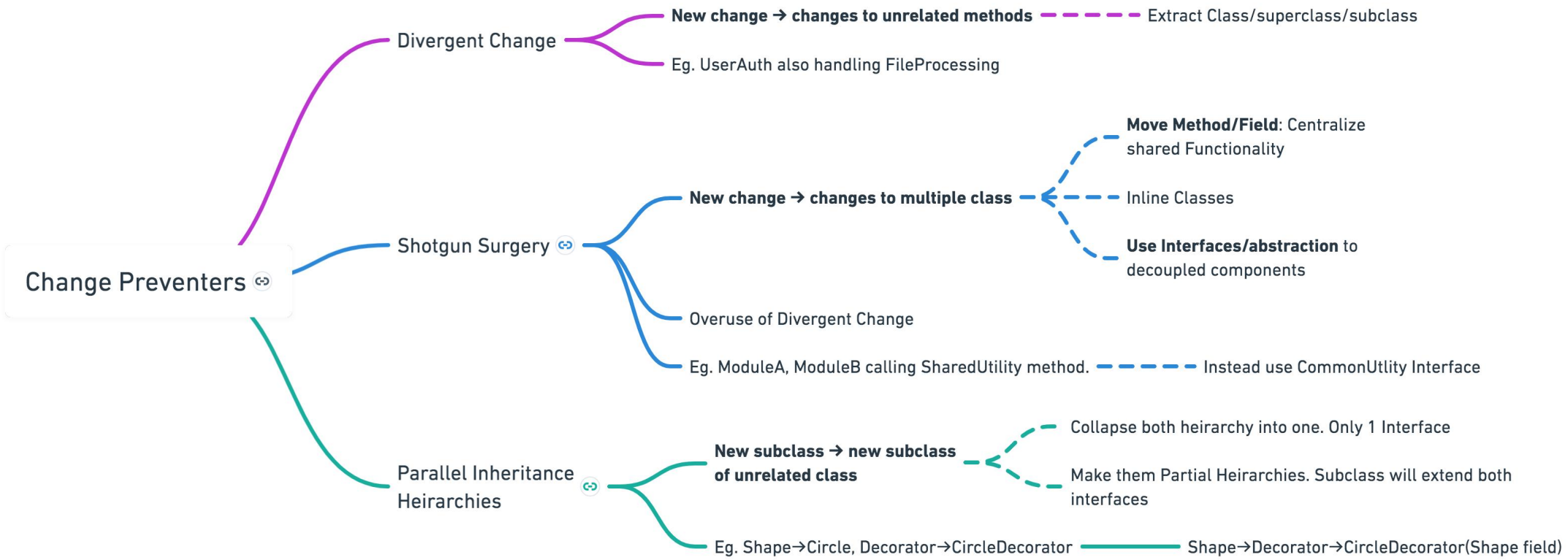
```
// Before
if (conditionA) {
    // logic for conditionA
    if (conditionB) {
        // logic for conditionB
    }
} else {
    // logic for !conditionA
    if (conditionC) {
        // logic for conditionC
    }
}

// After
void handleConditionA() {
    // logic for conditionA
}

void handleConditionB() {
    // logic for conditionB
}

void handleConditionC() {
    // logic for conditionC
}

if (conditionA) {
    handleConditionA();
    if (conditionB) {
        handleConditionB();
    }
} else {
    // logic for !conditionA
    handleConditionC();
}
```

```
// Before
if (userIsAdmin && userIsLoggedIn && userHasPermission) {
    // complex logic for admin actions
}

// After
if (!userIsAdmin) {
    // handle non-admin case
    return;
}

if (!userIsLoggedIn || !userHasPermission) {
    // handle lack of permissions
    return;
}

// simplified logic for admin actions
```

Alt Class with diff Interfaces

```
// Before
class DatabaseConnection {
    void connectToDatabase() {
        // logic for connecting to a database
    }
}

class DataConnector {
    void establishConnection() {
        // logic for establishing a connection
    }
}

// After
interface Connection {
    void connect();
}

class DatabaseConnection implements Connection {
    @Override
    public void connect() {
        // logic for connecting to a database
    }
}

class DataConnector implements Connection {
    @Override
    public void connect() {
        // logic for establishing a connection
    }
}
```

```
interface Connection {
    void connect();
}

class DatabaseConnection {
    void connectToDatabase() {
        // logic for connecting to a database
    }
}

class DataConnectorAdapter implements Connection {
    private DataConnector dataConnector = new DataConnector();

    @Override
    public void connect() {
        dataConnector.establishConnection();
    }
}
```


Change Preventers
Divergent Change
New change → changes to unrelated methods
Extract Class/superclass/subclass
Eg. UserAuth also handling FileProcessing
Shotgun Surgery
New change → changes to multiple class
Move Method/Field: Centralize shared Functionality
Inline Classes
Use Interfaces/abstraction to decoupled components
Overuse of Divergent Change
Eg. ModuleA, ModuleB calling SharedUtility method.
Instead use CommonUtlity Interface
Parallel Inheritance Heirarchies
New subclass → new subclass of unrelated class
Collapse both heirarchy into one. Only 1 Interface
Make them Partial Heirarchies. Subclass will extend both interfaces
Eg. Shape→Circle, Decorator→CircleDecorator
Shape→Decorator→CircleDecorator(Shape field)

```java
// Before
public class AuthAndFileHandler {
    public void authenticateUser() {
        // Authentication logic
    }

    public void handleFile() {
        // File handling logic
    }
}

// After
public class AuthHandler {
    public void authenticateUser() {
        // Authentication logic
    }
}

public class FileHandler {
    public void handleFile() {
        // File handling logic
    }
}
```

Divergent Change

```java
// Before
public class SharedUtility {
    public void doSomething() {
        // Shared logic
    }
}

public class ModuleA {
    private SharedUtility utility = new SharedUtility();

    public void performTask() {
        utility.doSomething();
    }
}

public class ModuleB {
    private SharedUtility utility = new SharedUtility();

    public void executeAction() {
        utility.doSomething();
    }
}
```

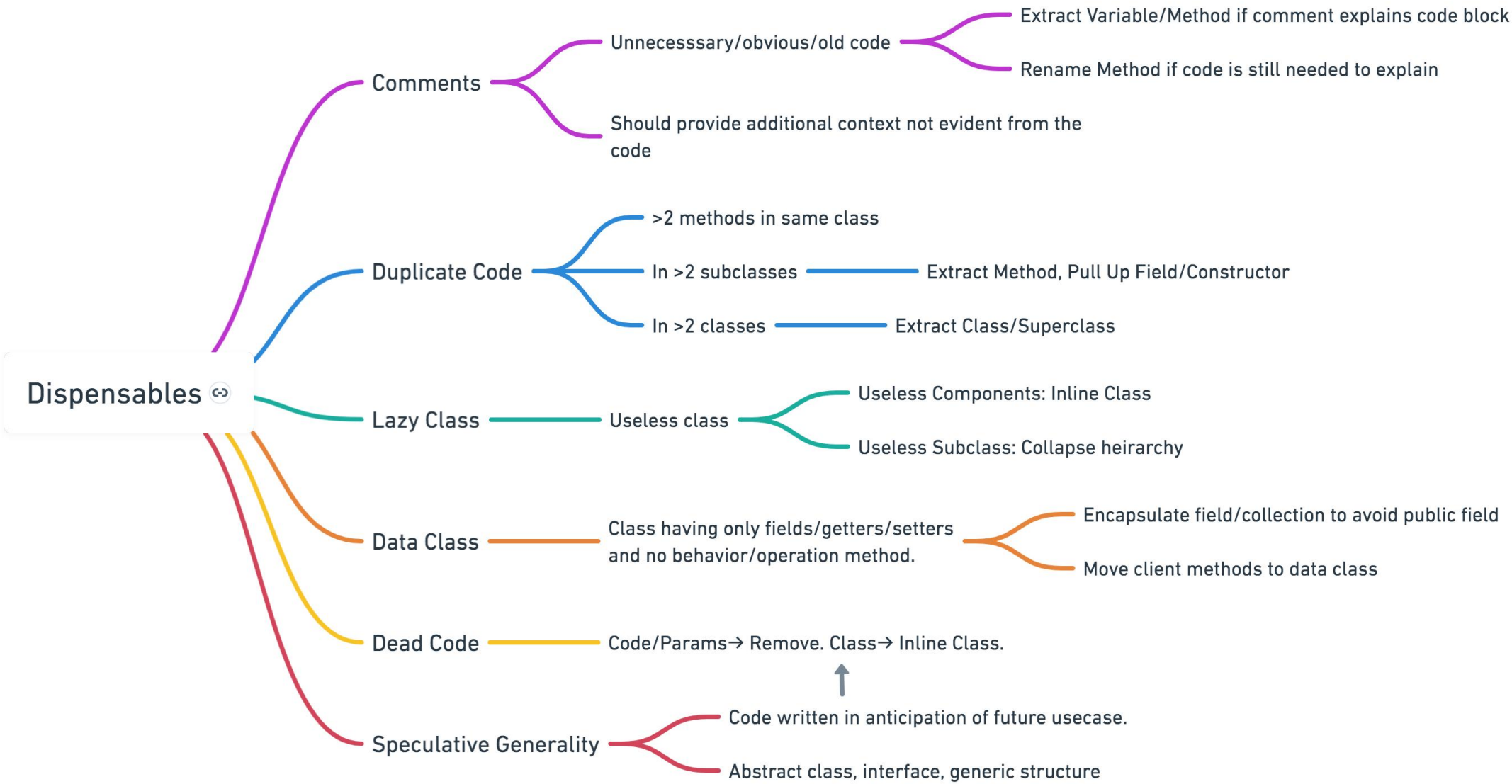
```java
// After
public interface CommonFunctionality {
    void doSomething();
}

public class SharedUtility implements CommonFunctionality {
    public void doSomething() {
        // Shared logic
    }
}

public class ModuleA {
    private CommonFunctionality functionality;

    // Constructor injection or setter method
    public ModuleA(CommonFunctionality functionality) {
        this.functionality = functionality;
    }

    public void performTask() {
        functionality.doSomething();
    }
}

public class ModuleB {
    private CommonFunctionality functionality;

    // Constructor injection or setter method
    public ModuleB(CommonFunctionality functionality) {
        this.functionality = functionality;
    }

    public void executeAction() {
        functionality.doSomething();
    }

    // ...
```

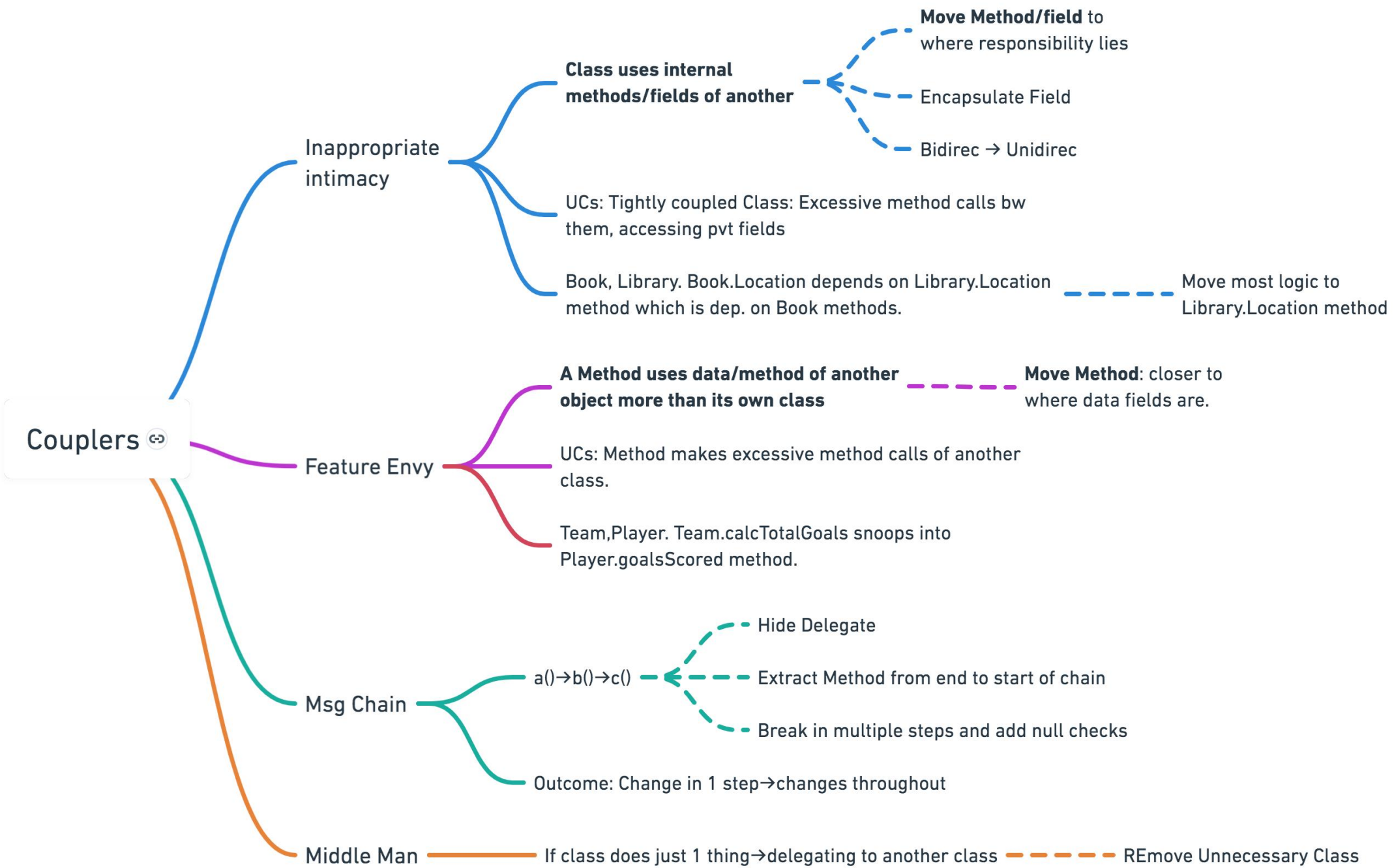
ShotGun Surgery

```java
// Before
public class Circle {
    // Circle-specific logic
}

public class ColoredCircle extends Circle {
    // ColoredCircle-specific logic
}

public interface ShapeDecorator {
    void decorate();
}

public class ColoredCircleDecorator extends ColoredCircle implements Shap
    // ColoredCircleDecorator-specific logic
    public void decorate() {
        // Decoration logic
    }
}
```

```java
// After
public interface Shape {
    void draw();
}

public class Circle implements Shape {
    public void draw() {
        // Circle drawing logic
    }
}

public interface ShapeDecorator extends Shape {
    void decorate();
}

public class ColoredCircleDecorator implements ShapeDecorator {
    private Shape decoratedShape;

    public ColoredCircleDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    public void draw() {
        decoratedShape.draw();
        decorate();
    }

    public void decorate() {
        // Decoration logic
    }
}
```

Parallel Inherritance Heirarchy


Dispensables
Comments
Unnecesssary/obvious/old code
Extract Variable/Method if comment explains code block
Rename Method if code is still needed to explain
Should provide additional context not evident from the code
Duplicate Code
>2 methods in same class
In >2 subclasses
Extract Method, Pull Up Field/Constructor
In >2 classes
Extract Class/Superclass
Lazy Class
Useless class
Useless Components: Inline Class
Useless Subclass: Collapse heirarchy
Data Class
Class having only fields/getters/setters and no behavior/operation method.
Encapsulate field/collection to avoid public field
Move client methods to data class
Dead Code
Code/Params→ Remove. Class→ Inline Class.
Speculative Generality
Code written in anticipation of future usecase.
Abstract class, interface, generic structure


Couplers
Inappropriate intimacy
Class uses internal methods/fields of another
Move Method/field to where responsibility lies
Encapsulate Field
Bidirec → Unidirec
UCs: Tightly coupled Class: Excessive method calls bw them, accessing pvt fields
Book, Library. Book.Location depends on Library.Location method which is dep. on Book methods.
Move most logic to Library.Location method
Feature Envy
A Method uses data/method of another object more than its own class
Move Method: closer to where data fields are.
UCs: Method makes excessive method calls of another class.
Team,Player. Team.calcTotalGoals snoops into Player.goalsScored method.
Msg Chain
a()→b()→c()
Hide Delegate
Extract Method from end to start of chain
Break in multiple steps and add null checks
Outcome: Change in 1 step→changes throughout
Middle Man
If class does just 1 thing→delegating to another class
REmove Unnecessary Class

```java
public class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }
}

public class Library {
    private List<Book> books;
    private String location;

    public Library(String location) {
        this.location = location;
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public boolean containsBook(Book book) {
        return books.contains(book);
    }

    public String getLocation() {
        return location;
    }

    // Updated method in Library to handle the association
    public String getLocationOfBook(Book book) {
        if (containsBook(book)) {
            return location;
        } else {
            return "Location not found";
        }
    }
```

```java
public class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    // Inappropriate Intimacy
    public String getLibraryLocation(Library library) {
        return library.getLocation(this);
    }
}

public class Library {
    private List<Book> books;
    private String location;

    public Library(String location) {
        this.location = location;
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    // Inappropriate Intimacy
    public String getLocation(Book book) {
        for (Book b : books) {
            if (b.equals(book)) {
                return location;
            }
        }
        return "Location not found";
    }
```

Inappropriate Intimacy

```java
public class UserDetails {
    public String username;
    public String password;

    // Constructors and methods...
}
```

```java
public class UserDetails {
    private String username;
    private String password;

    public UserDetails(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    // No direct exposure of the password
}
```

Indecent Exposure

```java
import java.util.List;

class Author {
    private String name;

    public String getName() {
        return name;
    }
}

class Book {
    private Author author;

    public Author getAuthor() {
        return author;
    }
}

class Library {
    private List<Book> books;

    public List<Book> getBooks() {
        return books;
    }
}

// Somewhere in the code...
Library library = getLibrary();
String authorName = library.getBooks().get(0).getAuthor().getName();
```

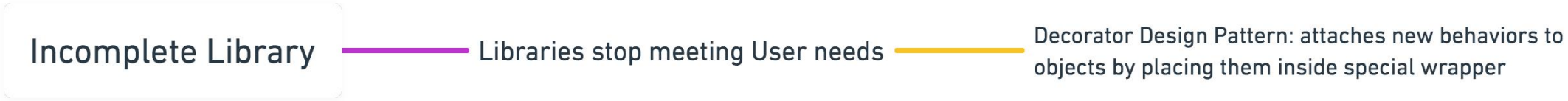
```java
Library library = getLibrary();

if (library == null || library.getBooks().isEmpty()) {
    throw new IllegalStateException("No books found in the library");
}

Book firstBook = library.getBooks().get(0);

if (firstBook == null || firstBook.getAuthor() == null) {
    throw new IllegalStateException("No author found for the first boo
}

String authorName = firstBook.getAuthor().getName();
```

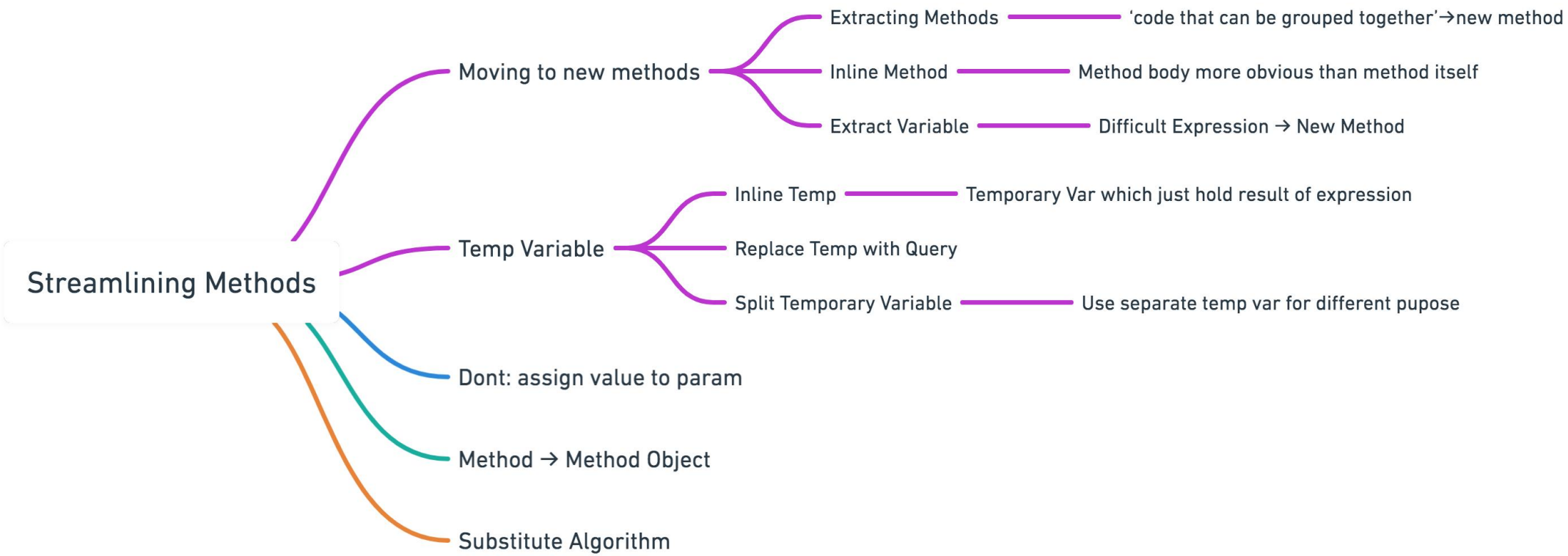
Message Chain

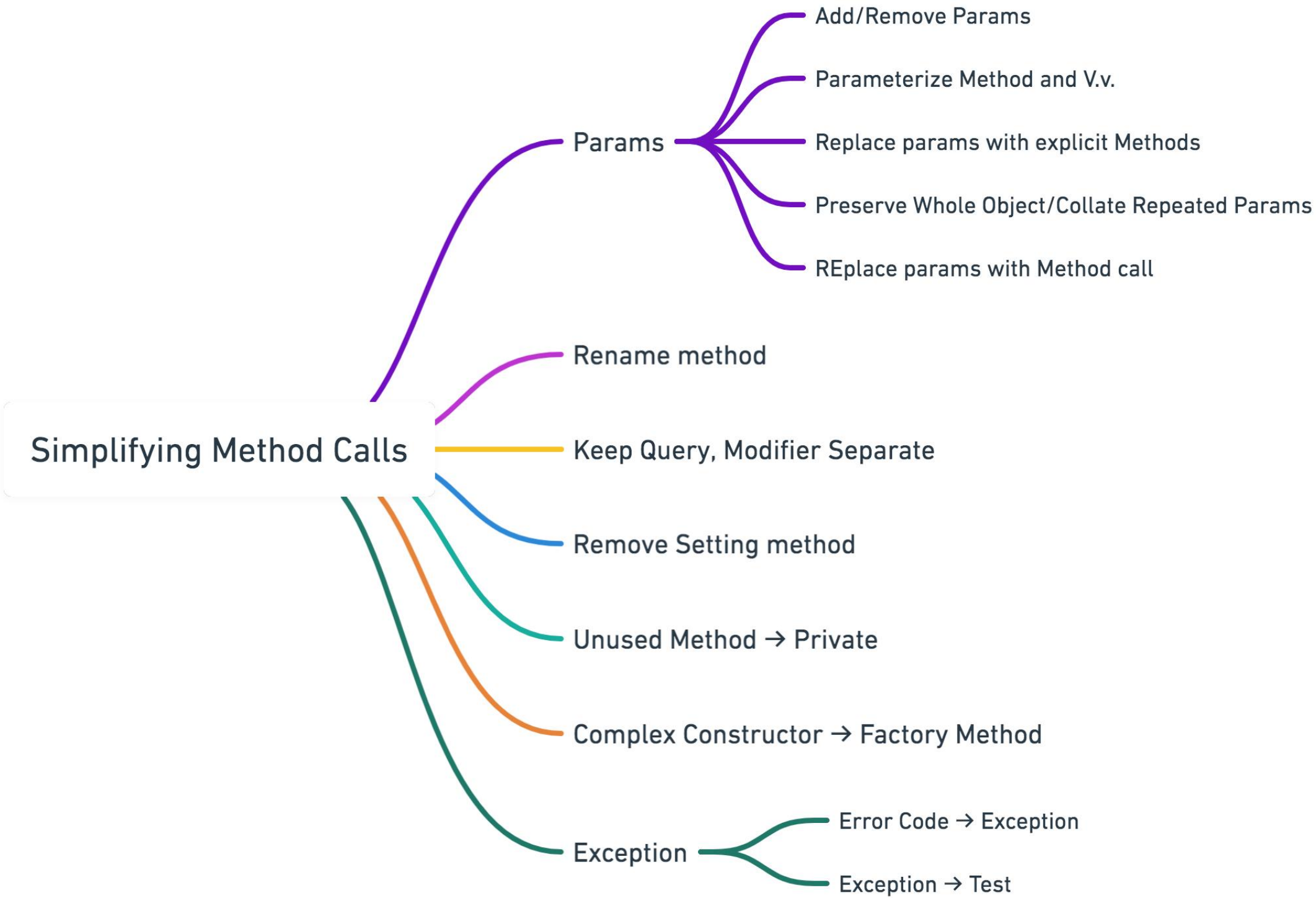
```java
public class NotificationService {
    public void sendNotification(User user, String message) {
        // Implementation details for sending a notification to a use
    }
}

public class User {
    private String username;

    public User(String username) {
        this.username = username;
    }

    public String getUsername() {
        return username;
    }
}

public class Messenger {
    private NotificationService notificationService;

    public Messenger(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    // Unnecessary middle man method
    public void sendMessage(User user, String message) {
        notificationService.sendNotification(user, message);
    }
}

// Somewhere in the code...
NotificationService notificationService = new NotificationService();
Messenger messenger = new Messenger(notificationService);
User user = new User("Alice");

// Middle Man usage
messenger.sendMessage(user, "Hello, Alice! You've got a message.");
```

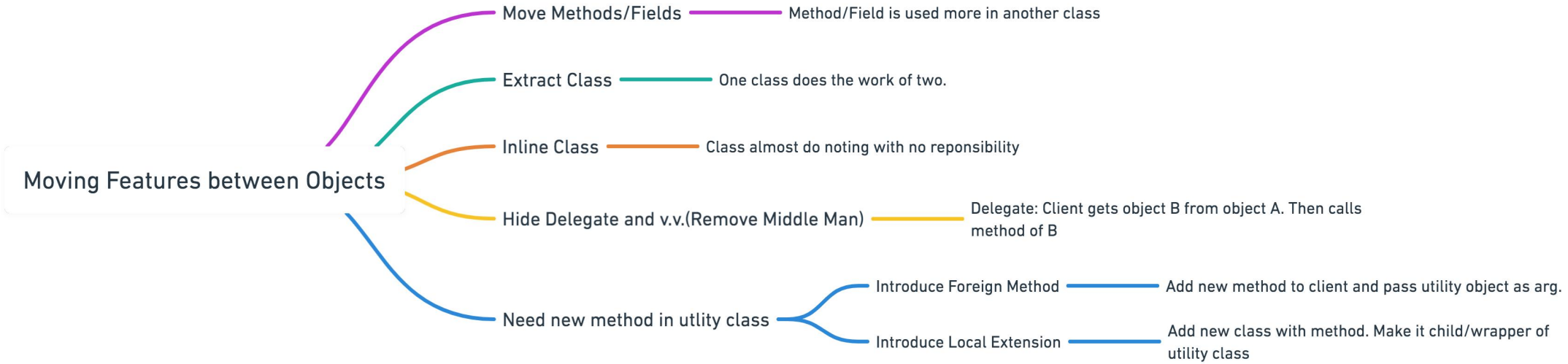
```java
// Somewhere in the code...
NotificationService notificationService = new NotificationService();
User user = new User("Alice");

// Refactored without middle man
notificationService.sendNotification(user, "Hello, Alice! You've got
```

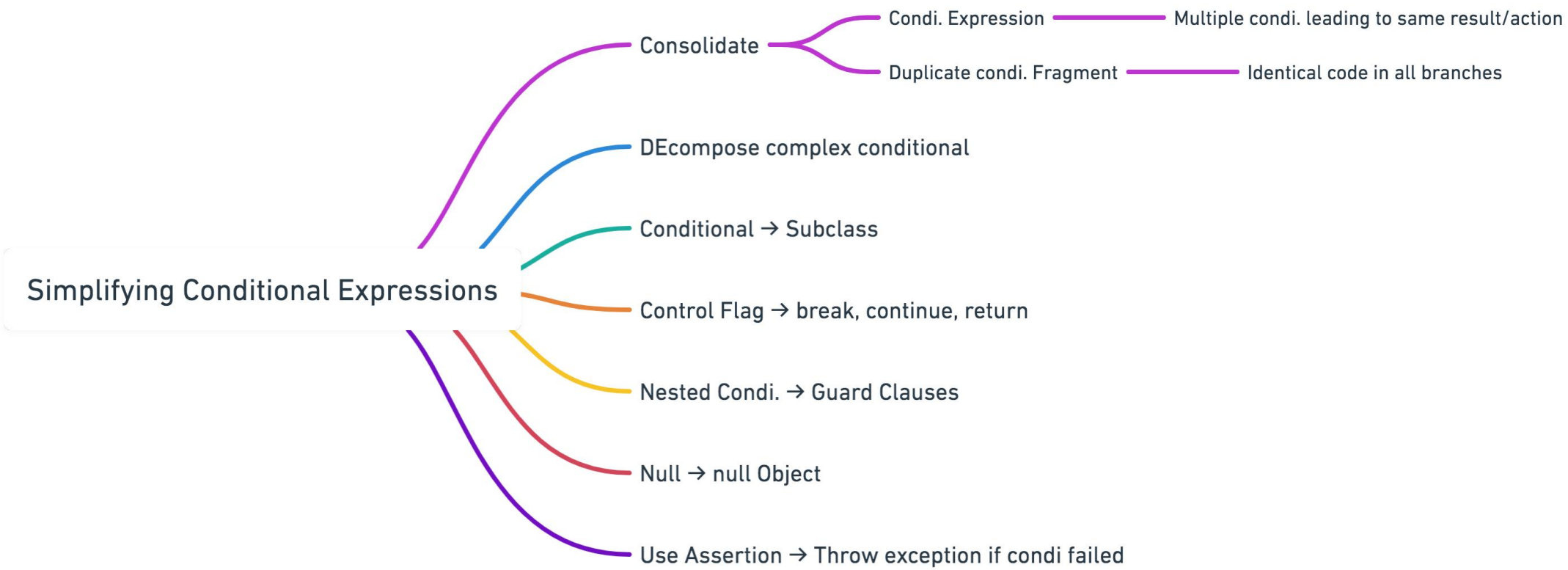
Middle Man

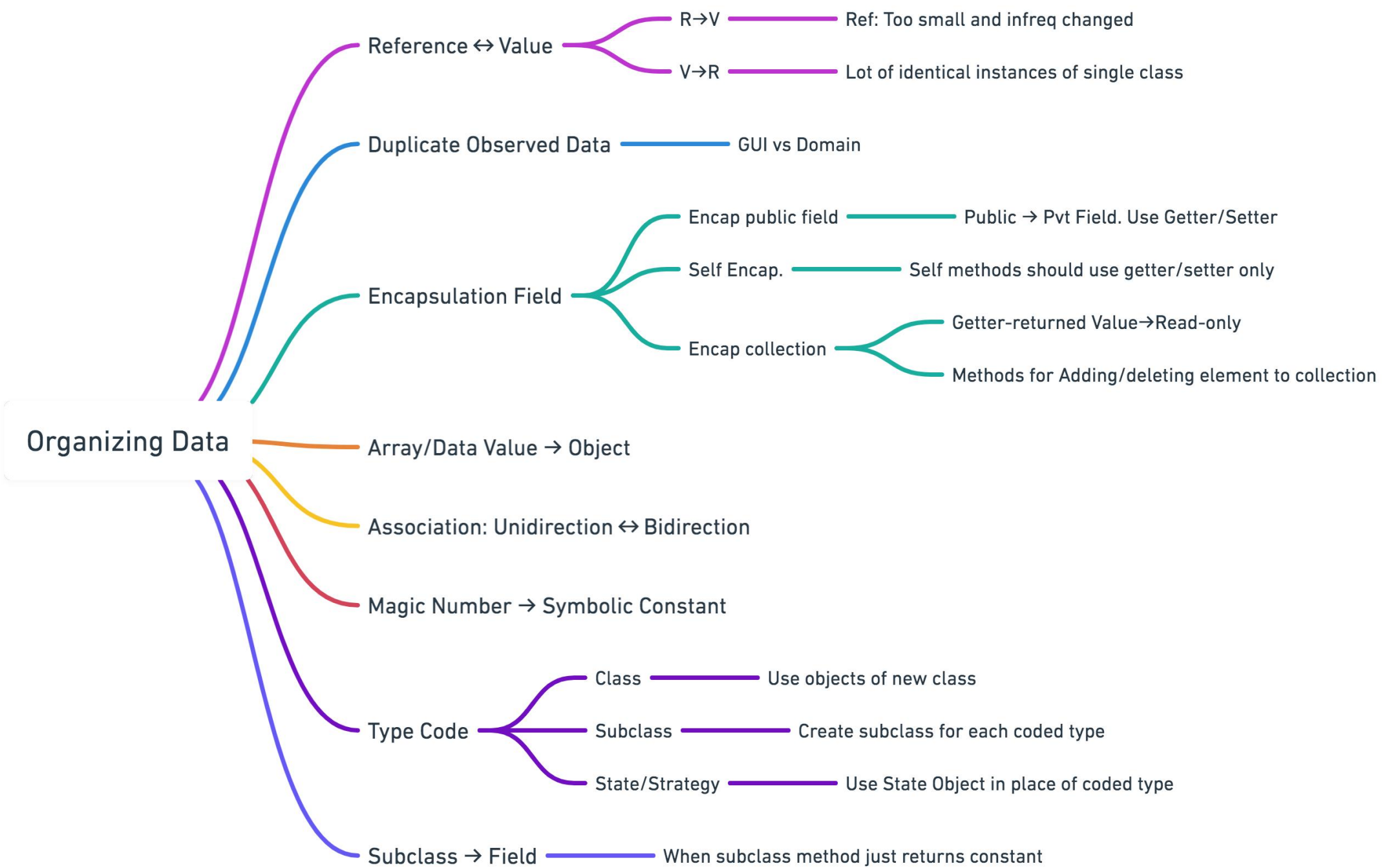

Incomplete Library
Libraries stop meeting User needs
Decorator Design Pattern: attaches new behaviors to objects by placing them inside special wrapper

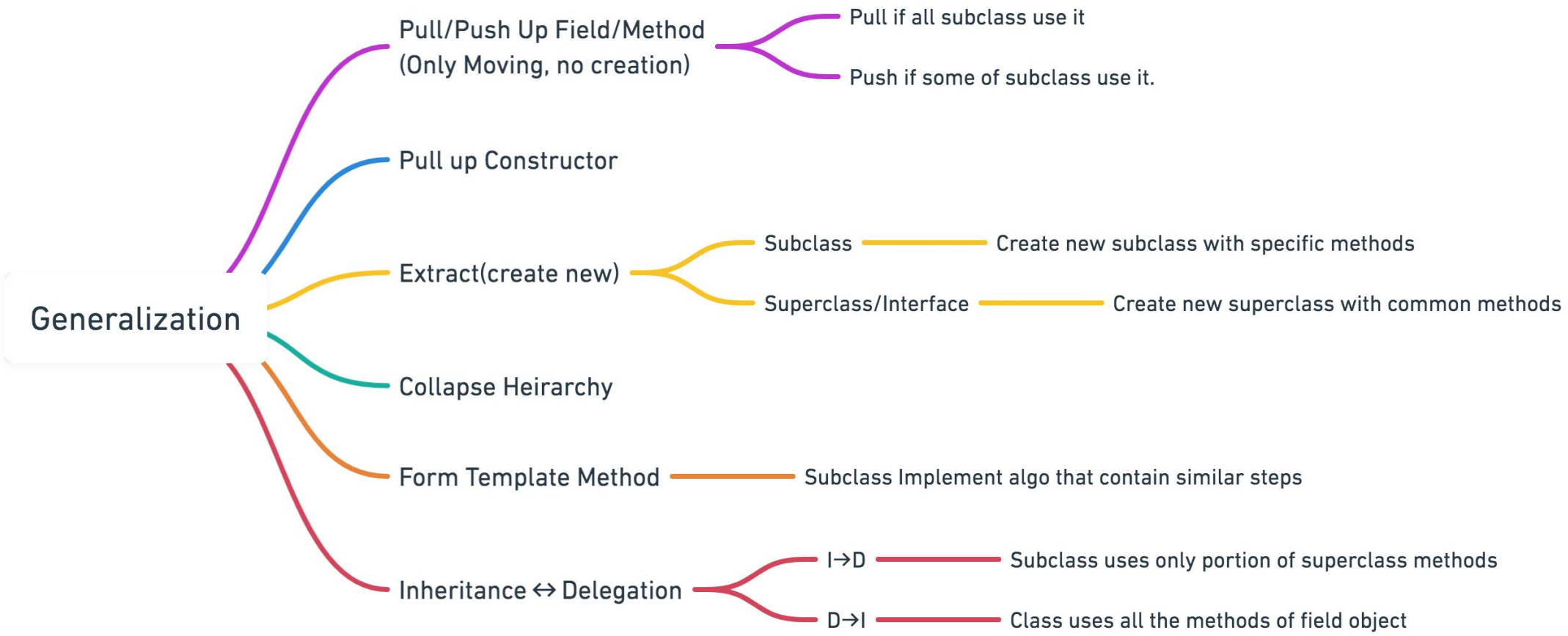

Streamlining Methods
Moving to new methods
Extracting Methods
'code that can be grouped together'→new method
Inline Method
Method body more obvious than method itself
Extract Variable
Difficult Expression → New Method
Temp Variable
Inline Temp
Temporary Var which just hold result of expression
Replace Temp with Query
Split Temporary Variable
Use separate temp var for different pupose
Dont: assign value to param
Method → Method Object
Substitute Algorithm


Simplifying Method Calls
Params
Add/Remove Params
Parameterize Method and V.v.
Replace params with explicit Methods
Preserve Whole Object/Collate Repeated Params
REplace params with Method call
Rename method
Keep Query, Modifier Separate
Remove Setting method
Unused Method → Private
Complex Constructor → Factory Method
Exception
Error Code → Exception
Exception → Test


Moving Features between Objects
Move Methods/Fields
Method/Field is used more in another class
Extract Class
One class does the work of two.
Inline Class
Class almost do noting with no reponsibility
Hide Delegate and v.v.(Remove Middle Man)
Delegate: Client gets object B from object A. Then calls method of B
Need new method in utlity class
Introduce Foreign Method
Add new method to client and pass utility object as arg.
Introduce Local Extension
Add new class with method. Make it child/wrapper of utility class


Simplifying Conditional Expressions
Consolidate
Condi. Expression
Multiple condi. leading to same result/action
Duplicate condi. Fragment
Identical code in all branches
DEcompose complex conditional
Conditional → Subclass
Control Flag → break, continue, return
Nested Condi. → Guard Clauses
Null → null Object
Use Assertion → Throw exception if condi failed


Organizing Data
Reference ↔ Value
R→V
Ref: Too small and infreq changed
V→R
Lot of identical instances of single class
Duplicate Observed Data
GUI vs Domain
Encapsulation Field
Encap public field
Public → Pvt Field. Use Getter/Setter
Self Encap.
Self methods should use getter/setter only
Encap collection
Getter-returned Value→Read-only
Methods for Adding/deleting element to collection
Array/Data Value → Object
Association: Unidirection ↔ Bidirection
Magic Number → Symbolic Constant
Type Code
Class
Use objects of new class
Subclass
Create subclass for each coded type
State/Strategy
Use State Object in place of coded type
Subclass → Field
When subclass method just returns constant


Generalization
Pull/Push Up Field/Method
(Only Moving, no creation)
Pull if all subclass use it
Push if some of subclass use it.
Pull up Constructor
Extract(create new)
Subclass
Create new subclass with specific methods
Superclass/Interface
Create new superclass with common methods
Collapse Heirarchy
Form Template Method
Subclass Implement algo that contain similar steps
Inheritance ↔ Delegation
I→D
Subclass uses only portion of superclass methods
D→I
Class uses all the methods of field object