JAVA-201

Code Smells

Parallel Inheritance Hierarchy

Every time you add a class to one package, you have to add a class to another package.

- Customer → CustomerService
- Order → OrderService
- Product → ProductService

Solution: **Cohesion - Move code from one hierarchy to the other, and then collapse what's left into fewer classes.**

Pattern: Rich Domain Model

Parallel Inheritance Hierarchy

```
class Customer {}
class CustomerService {}

class Order {}
class OrderService {}

class Product {}
class ProductService {}
```

Parallel Inheritance Hierarchy

Solution: Cohesion - Move code from one hierarchy to the other, and then collapse what's left into fewer classes.

```
class Customer {}
class Order {}
class Product {}

class Service {
void process(Customer customer) {...}
void process(Order order) {...}
void process(Product product) {...}}
```

Lazy Class, Middle Man & Dead Code

Lazy Class: Doesn't do much to justify its existence.

Middle Man: Delegates to another class but doesn't add its own value.

Dead Code: Code no longer used.

Lazy Class, Middle Man & Dead Code
Solutions

Any code that exists takes maintenance cost. Eliminate code that doesn't justify its existence. Apply Cohesion.

Lazy Class: **Inline** its behavior into other classes.

Middle Man: **Combine** the Middle Man with the class to which it delegates.

Dead Code: **Delete** it.

Lazy Class, Middle Man & Dead Code

```
class MiddleMan {
      void process() {
            Worker.doWork();
       }
}
```

```
class Worker {
      void doWork() {
      System.out.println ("Work done");
    }
}
```

Alternative Classes with Different Interfaces

Classes that are similar but are not substitutable, since they do not share the same supertype.

```
/** Stores & retrieves customer info to/from the file system. **/
class CustomerFileData {
   void save(Customer customer) {...}
   Customer getByCustId(int id) {...}
   Collection<Customer> getCustomersByLastName {...}
...
}
/** Stores & retrieves customer info to/from RDBMS. **/
class CustomerDAO {
   void saveOrUpdate(Customer customer) {...}
   Customer findById(int id) {...}
   Collection<Customer> findByLastName {...}
...
}
```

Alternative Classes with Different Interfaces

Choices have to be hard-coded, or messy conditionals need to be used.

```
if (...) { // save to file
   new CustomerFileData().save(customer);
} else if (...) { // save to RDBMS
   new CustomerDAO().saveOrUpdate(customer);
} ...
```

What if we have to support other forms of data access (web services?) later on?

Alternative Classes with Different Interfaces

Solution: **Polymorphism** - Have the classes share a common supertype.

```
interface CustomerRepository {
   void save(Customer customer);
   Customer findById(int id);
   Collection<Customer> findByLastName;
...
}
class CustomerFileData implements CustomerRepository {
...
}
class CustomerDAO implements CustomerRepository {
...
}
```

Alternative Classes with Different Interfaces

Solution: **Polymorphism** - Have the classes share a common supertype.

```
private CustomerRepository repo;

void setCustomerRepository(CustomerRepository repo) {
   this.repo = repo;
}
...
       repo.save(customer);
...
```

Patterns: Strategy, State,

Data Class or Anemic Domain Model

Class that only contains data and not behavior.

```
public class BankAccount {
   private int acctNo;
   private BigDecimal balance;

   public void setAcctNo(int acctNo) {
       this.acctNo = acctNo;
   }

   public int getAcctNo() {
       return acctNo;
   }

   public void setBalance(BigDecimal balance) {
       this.balance = balance;
   }

   public BigDecimal getBalance() {
       return balance;
```

Data Class or Anemic Domain Model

- Balance should not be negative.
- ATM withdrawals should not exceed 20,000.
- All transactions must be logged with a transaction ID.

In an Anemic Domain Model, all this behavior needs to be done outside the object. Error-prone.

Object may be in an invalid state (ex. negative balance). Constant validation must be done.

Fields can be changed by any code. Encapsulation is broken.

Data Class or Anemic Domain Model

Solution: Implement a **Rich Domain Model**. Implement **Information Expert**. Place methods in the class with the data they operate on.

- Remove unneeded getters and setters, to preserve encapsulation.

```
public class BankAccount {
   private final int acctNo;
   private BigDecimal balance;

   public BankAccount(int acctNo) {...}

   public Transaction deposit(BigDecimal amt) {
       if (amt.signum() < 0) {
           throw new IllegalArgumentException("cannot be negative");
       }
   ...
   }
```

Data Class or Anemic Domain Model

```
...
    public Transaction withrawal(BigDecimal amt) {
        if (amt.compareTo(balance) > 0) {
            throw new WithrawalExceedsBalanceException(...);
        }
    ...
    }

    public Transaction withdrawalViaAtm(BigDecimal amt) {
        if (amt.compareTo(MAX_ATM_WITHDRAWAL) > 0) {
            throw new MaxAtmWithdrawalExceededException(...);
        }
    ...
    }
}
```

Refused Bequest

When a class inherits members that it doesn't need.

- Ok if the unneeded members are encapsulated.
- Dangerous if the unneeded members are part of the interface.

Solution: Replace inheritance with **composition**.

Refused Bequest

```
class Animal {
        void walk() {}
         void fly() {}

}

class Dog extends Animal  {
// Only uses walk, not fly

}
```

Refused Bequest

- Solution: The solution is to use composition instead of inheritance, allowing classes to include only the behaviors they need.

```
interface Walker   {
       void walk();
}

class Dog implements Walker {
      public void walk() {
      System.out.println("Dog is walking");    }}
```

Speculative Generality

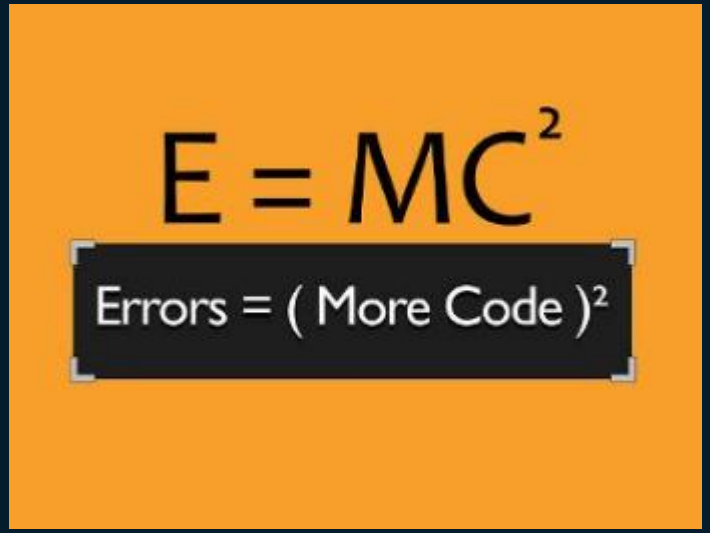
When code is written to be more flexible than the present requirements need it to be.

Usually in **anticipation of future requirements.**

Speculative Generality

- Methods that try to accommodate any data type.
  - ex. String or int instead of enum, LocalDate, BigDecimal or custom class
- Overloaded methods/constructors that are not or rarely used.
- Methods that don't do much, and don't add to clarity.
- Parameters that aren't used.
- Supertypes that only have one subtype.
- Middle Man, Lazy Class, Dead Code

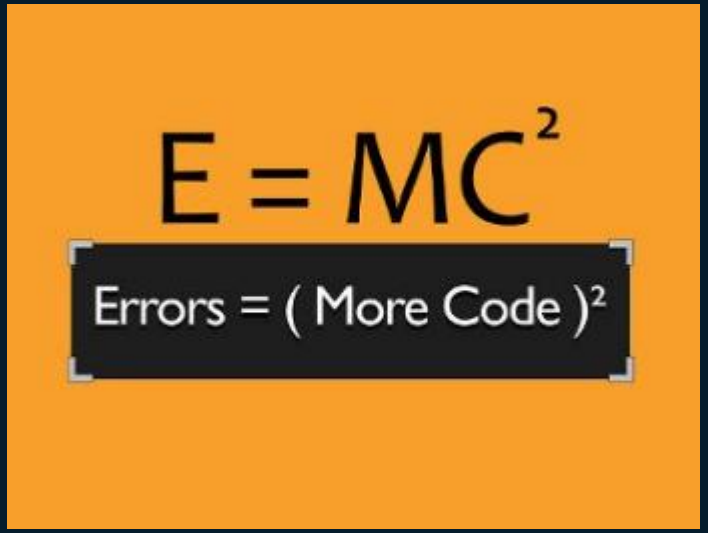
Code is hard to understand and maintain.



Easy to make code more complex when needed. Very hard to make code simpler!

Speculative Generality

```
class DataProcessor {
        void process(String data) {...}
        void process(int data) {...}
        void process(Object data) {...} //
Unnecessary over-generalization}
```



Speculative Generality
Solution

Code just enough to satisfy the requirements of the current Sprint.

YAGNI - You Ain't Gonna Need It
DTSTTCPW - Do the Simplest Thing That Could Possibly Work

Requirements change. Even if the written requirements state that something is required, that might change later, and any effort you do to fulfilling those requirements will be wasted.

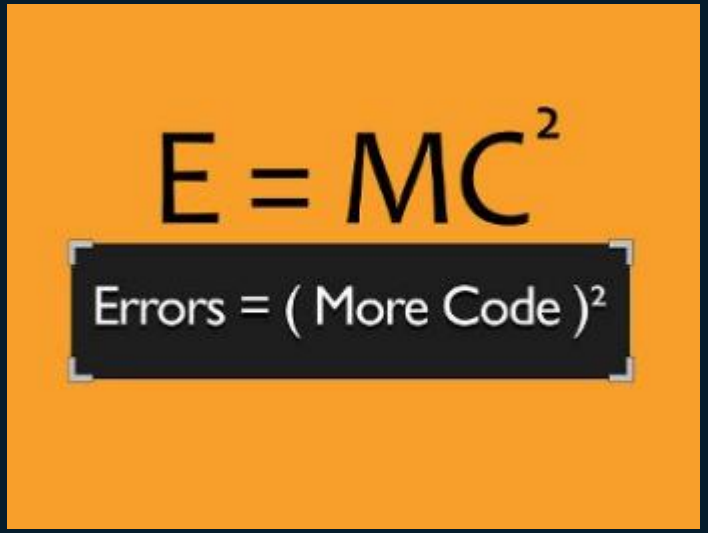
Focus on the more certain requirements - those in the current Sprint.

- Code is simpler, more readable, less likely to be buggy, gets done on-time.

Collapse unneeded hierarchies, inline methods and classes, remove unused parameters, etc.

Speculative Generality

```
class DataProcessor {
        void process(String data) {...}
        void process(int data) {...}
        void process(Object data) {...} //
Unnecessary over-generalization}
```




```
class DataProcessor {
void process(String data) {...}}
```

God Class / God Method

Much of the code is concentrated in one class or method, or many other classes or methods depend on that one class or method.

Solution:

- Single Responsibility Principle
- Information Expert
- Law of Demeter

God Class / God Method

```
class GodClass {
        void handleEverything() {
                processOrders();
                managePayments();
                sendNotifications();
                 generateReports();
          }
}
```

```
class OrderHandler   {
        void processOrders() {...}
}

class PaymentHandler {
        void managePayments() {...}
}

class NotificationHandler  {
         void sendNotifications() {...}
}
```

Temporary Field

Class or instance fields that are temporarily set for method operations. They don't describe the state of the object.

```
class Calculation {
   private int tempValue;

   void calculate() {
        tempValue = 5;
       tempValue += 10;
   }
}
```

Dangerous in multithreaded environment! Multiple threads may be modifying the field in the middle of method executions.

Confusing for other teammates (or your future self) since it's hard to figure out what the field is for.

Temporary Field
Solutions

- Make the field/s local variable/s.

```
class Calculation {
   void calculate()  {
        int tempValue = 5;
       tempValue += 10;
    }
}
```

- If there are several of them and they end up getting passed around together as parameters, encapsulate them in a new class.

Message Chains or Train Wrecks

Reaching into an object's attributes to call the attributes' methods.

```
currSection.getSchedule().equals(newSection.getSchedule())
```

Causes tight coupling. The calling class becomes tightly coupled to the internal structure of the object. The object cannot change without breaking the caller.

Message Chains or Train Wrecks
Solution

Apply **Information Expert / Law of Demeter**

- Don't get the fields. Create a method that does the work and just get the results.

```
currSection.hasConflict(newSection)
```

No tight coupling. Internal structure of Section can change without breaking callers.

Easier to read & maintain.

Message Chains or Train Wrecks
Solution

Apply **Information Expert / Law of Demeter**

```
class Section {
      boolean hasConflict(Section newSection)    {

      return this.schedule.hasConflict(newSection.schedule);
      }
}
```

Message Chains or Train Wrecks

Another problem with message chains is that changes in one class can cascade into multiple other classes, forcing widespread modifications. This can be avoided by encapsulating behavior inside the appropriate classes.

```
customer.getOrder().getShippingInfo().getAddress().getCity();
```

Message Chains or Train Wrecks

Instead of chaining method calls like this, refactor by introducing a method in Customer to retrieve the needed information directly:

```
class Customer {
      String getShippingCity()    {
            return order.getShippingCity();
       }
}

class Order {
        String getShippingCity()   {
              return shippingInfo.getCity();
}

}
            customer.getShippingCity();
```

"How" Comments

If you need a comment to understand how some code works, then the code is not readable:

```
// start of fund transfer process
// end of fund transfer process
// check if section schedule has
conflict with existing section
```

"How" Comments

"Why" comments are good:

```
// This method uses encryption library Xxx to comply with US
regulations on...
// This field is immutable in order to make it threadsafe.
// This private no-arg constructor is to allow JPA to set the
final fields.
```

Javadoc comments that state the contract that overridden methods need to keep are also good.

"How" Comments
Solution: Make the code itself readable.

Extract Method where you find a comment.

```
// start of fund transfer process
↓
account.fundTransfer();

// check if section schedule has conflict with existing sections
↓
section.hasConflict(otherSection)
```

"How" Comments
Solution: Make the code itself readable.

Replace Magic Numbers with **Explaining Variables**:

```
if (sections.size() >= 6) { // max enrollable sections
    ...
}

↓

final int MAX_ENROLLABLE_SECTIONS = 6;
if (sections.size() >= MAX_ENROLLABLE_SECTIONS) {
    ...
}
```

"How" Comments
Solution: Make the code itself readable.

If a variable or method still needs a "How" comment, rename the variable or method:

```
// # of people needed to approve document
private int constraint;
↓
private int numberOfApprovers;

// enlist the student in the section
student.add(section);
↓
student.enlist(section);
```

There are many other code smells being discovered and documented all the time.

Continue to read object-oriented design articles and books to learn more.