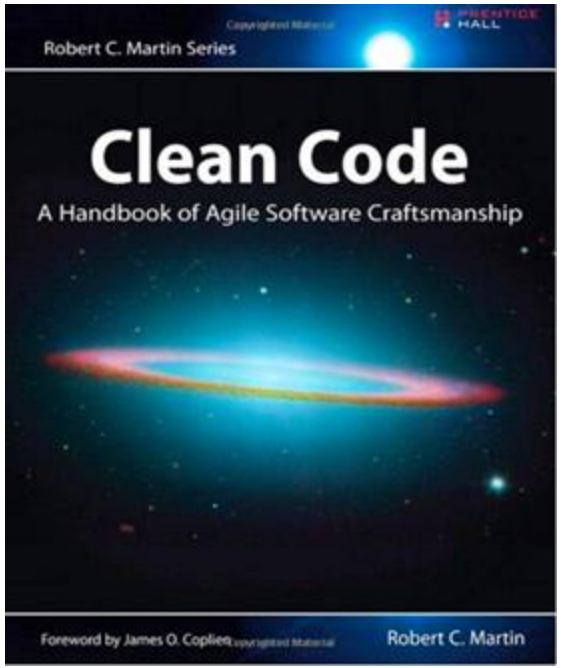
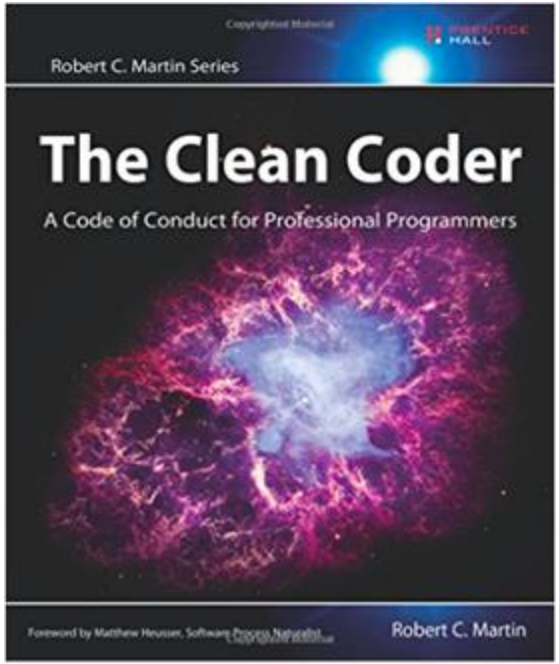
Clean Code and Layer Boundaries

We are learning from here:





Clean Code - General rules

- Follow standard conventions.
- Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible. (KISS)
- Boy scout rule. Leave the campground cleaner than you found it.
- Always find root cause. Always look for the root cause of a problem.

Clean Code - Design rules 1

Keep configurable data at high levels.

Will improve:

- **Flexibility:** Configurable data at high levels allows easy behavior modification, adapting to various environments without changing code.
- **Maintainability:** Separating configuration from code simplifies maintenance, clarifies responsibilities, and enables quicker updates.
- **Testability:** Independent configurable data enhances testing, enabling diverse scenarios by altering configurations alone.
- **Reusability:** Isolating configurable data fosters modular, reusable code, reducing development effort across projects.

Clean Code - Design rules 1(Cont.)

Keep configurable data at high levels.

To apply:

- **Command line arguments:** Pass configuration data as command line arguments when running the application.
- **Configuration files:** Store configuration data in external files, such as JSON, XML, or YAML, and read them when the application starts.
- **Environment variables:** Use environment variables to store configuration data, which can be easily set and accessed by the application.

Clean Code - Design rules 1(Cont.)

Keep configurable data at high levels.

- **Configuration File** (db-config.properties):

```
db.url=jdbc:mysql://localhost:3306/mydatabase
db.user=root
db.password=secret
```

Clean Code - Design rules 1(Cont.)

Keep configurable data at high levels.

- **Configuration Class** (Config.java):

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class Config {
    private final Properties properties;

    public Config(String configFilePath) throws IOException {
        properties = new Properties();
        properties.load(new FileInputStream(configFilePath));
    }

    public String getDatabaseUrl() {
        return properties.getProperty("db.url");
    }

    public String getDatabaseUser() {
        return properties.getProperty("db.user");
    }

    public String getDatabasePassword() {
        return properties.getProperty("db.password");
    }
}
```

Clean Code - Design rules 1(Cont.)

Keep configurable data at high levels.

- **Database Connection Class** (DatabaseConnector.java):

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnector {
    private final String url;
    private final String user;
    private final String password;

    public DatabaseConnector(String url, String user, String password) {
        this.url = url;
        this.user = user;
        this.password = password;
    }

    public Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url, user, password);
    }
}
```

Clean Code - Design rules 1(Cont.)

Keep configurable data at high levels.

- **Main Application** (Main.java):

```java
import java.io.IOException;
import java.sql.Connection;
import java.sql.SQLException;

public class Main {
    public static void main(String[] args) {
        try {
            Config config = new Config("path/to/db-config.properties");

            String url = config.getDatabaseUrl();
            String user = config.getDatabaseUser();
            String password = config.getDatabasePassword();

            DatabaseConnector dbConnector = new DatabaseConnector(url, user, password);

            try (Connection connection = dbConnector.getConnection()) {
                System.out.println("Connected to the database!");
                // Perform database operations here...
            } catch (SQLException e) {
                System.err.println("Failed to connect to the database.");
                e.printStackTrace();
            }
        } catch (IOException e) {
            System.err.println("Failed to load configuration.");
            e.printStackTrace();
        }
    }
}
```

Clean Code - Design rules 2

Prefer polymorphism to if/else or switch/case.

Will improve:

- **Cleaner code:** Polymorphism encapsulates behavior, improving code readability and comprehension.
- **Maintainability:** Polymorphism enables adding behaviors or classes without altering existing code, reducing error risk.
- **Flexibility:** Polymorphism supports the Open/Closed Principle, enhancing codebase flexibility by introducing new functionality without modifying existing code.

Clean Code - Design rules 2 (Cont.)

Prefer polymorphism to if/else or switch/case.

Polymorphism sticks to OCP helping in achieving this by allowing you to introduce new functionality with new classes rather t han modifying existing code.

To apply:

- Identify the common behavior or interface that different classes share.
- Create an abstract class or interface defining this common behavior.
- Make each class that shares the common behavior inherit from the abstract class or implement the interface.
- Replace the if/else or switch/case statements with code that leverages the common interface, allowing you to treat objects of different classes uniformly.

Clean Code - Design rules 2 (Cont.)

Prefer polymorphism to if/else or switch/case.

Bad example

```
1  public class ShapeDrawer {
2
3      public void drawShape(String shapeType) {
4          if ("circle".equals(shapeType)) {
5              System.out.println("Drawing a circle");
6          } else if ("rectangle".equals(shapeType)) {
7              System.out.println("Drawing a rectangle");
8          } else if ("triangle".equals(shapeType)) {
9              System.out.println("Drawing a triangle");
10         }
11         // ... additional shapes ...
12     }
13 }
```

Clean Code - Design rules 2 (Cont.)

Prefer polymorphism to if/else or switch/case.

Fix that

```java
public interface Shape {
    void draw();
}
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a triangle");
    }
}
```

Clean Code - Design rules 2 (Cont.)

Prefer polymorphism to if/else or switch/case.

Now it is better

```
1  public class ShapeDrawer {
2
3      public void drawShape(Shape shape) {
4          shape.draw();
5      }
6
7      public static void main(String[] args) {
8          ShapeDrawer shapeDrawer = new ShapeDrawer();
9
10         Shape circle = new Circle();
11         Shape rectangle = new Rectangle();
12         Shape triangle = new Triangle();
13
14         shapeDrawer.drawShape(circle);
15         shapeDrawer.drawShape(rectangle);
16         shapeDrawer.drawShape(triangle);
17     }
18 }
```

Clean Code - Design rules 3

Separate multi-threading code

Will improve:

- **Maintainability:** Isolating multi-threading code simplifies maintenance by focusing on concurrency issues separately from business logic.
- **Readability:** Separating multi-threading code enhances business logic understanding without distraction from concurrency complexity.
- **Testability:** Separating multi-threading code enables independent testing of business logic and concurrency, improving issue identification.

Clean Code - Design rules 3 (Cont.)

Separate multi-threading code

To apply:

- **Encapsulation:** Contain multi-threading code in dedicated classes or modules, separating it from business logic.
- **Abstraction:** Utilize high-level concurrency abstractions (e.g., thread pools, parallel loops, async/await) to reduce multi-threading complexity.

Clean Code - Design rules 3 (Cont.)

Separate multi-threading code

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class BusinessLogic {
    public void execute() {
        // Business logic here (no threading code)
        System.out.println("Executing business logic on thread: " + Thread.currentThread().getName());
    }
}

class ConcurrentExecutor {
    private final ExecutorService executorService = Executors.newFixedThreadPool(4);

    public void execute(Runnable task) {
        // This is where the multithreading code is
        executorService.submit(task);
    }

    public void shutdown() {
        executorService.shutdown();
    }
}

public class Main {
    public static void main(String[] args) {
        ConcurrentExecutor concurrentExecutor = new ConcurrentExecutor();
        BusinessLogic businessLogic = new BusinessLogic();

        // Execute business logic concurrently
        for (int i = 0; i < 10; i++) {
            concurrentExecutor.execute(businessLogic::execute);
        }

        concurrentExecutor.shutdown();
    }
}
```

Clean Code - Design rules 4

Prevent over configurability

Will cause:

- **Complexity:** Too many configuration options increase application complexity, hindering developers' understanding of codebase and component relationships.
- **Maintenance burden:** Excessive configuration options complicate maintenance and testing, as developers must account for multiple settings combinations.
- **User confusion:** Over-configurability overwhelms users, making it hard to discern essential options and optimal application configuration.

Clean Code - Design rules 4 (Cont.)

Prevent over configurability

To apply:

- **Prioritize essential options:** Focus on critical configuration options affecting application behavior; limit less important or seldom-used options.
- **Sensible defaults:** Offer sensible default values, enabling easy application usage with minimal configuration, while permitting customization as needed.
- **Group related options:** Organize configuration options into logical groups, simplifying user understanding and management.
- **Validate and document:** Thoroughly document configuration options, purposes, and valid values; implement validation to ensure acceptable limits.

Clean Code - Design rules 4(Cont.)

Prevent over configurability (too much)

```java
public class DataProcessor {

    private int bufferSize;
    private int threadCount;
    private String characterSet;
    private String endOfLineCharacters;
    private String dateFormat;
    // ... many more configurations ...

    public DataProcessor(int bufferSize, int threadCount, String characterSet,
                         String endOfLineCharacters, String dateFormat) {
        this.bufferSize = bufferSize;
        this.threadCount = threadCount;
        this.characterSet = characterSet;
        this.endOfLineCharacters = endOfLineCharacters;
        this.dateFormat = dateFormat;
        // ...
    }

    public void processData(String filePath) {
        // Logic to process data using the configurations
    }

    // ... getters and setters for all configurations ...
}
```

Clean Code - Design rules 4(Cont.)

Prevent over configurability (Better)

```java
public class DataProcessor {

    private static final int DEFAULT_BUFFER_SIZE = 8192; // 8 KB
    private static final String DEFAULT_CHARACTER_SET = "UTF-8";
    private static final String DEFAULT_END_OF_LINE = "\n";
    private static final String DEFAULT_DATE_FORMAT = "yyyy-MM-dd";

    private String dateFormat;

    public DataProcessor() {
        // Use sensible defaults
        this(DEFAULT_DATE_FORMAT);
    }

    public DataProcessor(String dateFormat) {
        this.dateFormat = dateFormat;
    }

    public void processData(String filePath) {
        // Logic to process data using the configurations
        // Use the default configurations for bufferSize, characterSet, etc.
        // Allow dateFormat to be configurable since it might be critical
    }
}
```

Clean Code - Design rules 5

Use Dependency Injection

Will improve:

- **Maintainability:** Dependency injection fosters loose coupling, simplifying dependency updates or replacements without impacting dependent classes.
- **Testability:** Dependency injection permits mock or stub dependency implementations for easier, isolated class testing.
- **Flexibility:** Dependency injection facilitates swapping dependency implementations or configurations, adapting to various environments or requirements.
- **Reusability:** Decoupling dependencies from implementation, dependency injection promotes code reusability and a modular design.

Clean Code - Design rules 5 (Cont.)

Use Dependency Injection

To Apply:

- **Identify dependencies:** Ascertain dependencies required for a class's functionality.
- **Refactor dependencies:** Modify the class to accept dependencies as input, avoiding direct instantiation (constructor, setter, interface injection).
- **Provide dependencies:** Instantiate and configure dependencies externally, passing them to dependent classes during instantiation or method invocation.
- **Use DI frameworks** (optional): Utilize available dependency injection frameworks or libraries (e.g., Spring) for efficient dependency management.

Clean Code - Design rules 5 (Cont.)

Use Dependency Injection (bad)

```java
public class EmailService {
    public void sendEmail(String message, String receiver) {
        // Logic to send email
        System.out.println("Email sent to " + receiver + " with message: " + message);
    }
}

public class Notification {
    private EmailService emailService = new EmailService();

    public void notifyUser(String message, String receiver) {
        emailService.sendEmail(message, receiver);
    }
}
```

Clean Code - Design rules 5 (Cont.)

Use Dependency Injection (better)

```java
public interface MessageService {
    void sendMessage(String message, String receiver);
}

public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message, String receiver) {
        // Logic to send email
        System.out.println("Email sent to " + receiver + " with message: " + message);
    }
}

public class Notification {
    private MessageService messageService;

    public Notification(MessageService messageService) {
        this.messageService = messageService;
    }

    public void notifyUser(String message, String receiver) {
        messageService.sendMessage(message, receiver);
    }
}
```

Clean Code - Design rules 6

Follow Law of Demeter. A class should know only its direct dependencies.

Meaning:

- An object should only call methods on: a. Itself b. Objects passed as method parameters c. Objects it creates or instantiates d. Component objects (objects directly held as instance variables)
- An object should not navigate through multiple levels of other objects or call methods on objects returned by other methods.

Clean Code - Design rules 6 (Cont.)

Follow Law of Demeter. A class should know only its direct dependencies.

Will improve:

- **Maintainability:** Loose coupling between classes makes it easier to update or replace parts of the system without affecting other components.
- **Readability:** The principal results in simpler interactions between objects, making the code more readable and easier to understand.
- **Flexibility:** The limited knowledge of other classes and their structures allows for greater flexibility when modifying or extending the code.

Clean Code - Design rules 6 (Cont.)

Follow Law of Demeter. A class should know only its direct dependencies.

To Apply:

- **Encapsulation:** Ensure that your classes expose only necessary properties and methods, keeping their internal structure and state hidden from other classes.
- **Delegation:** Instead of directly accessing properties or methods of other objects, delegate the responsibility to the object itself or use an intermediary object.
- **Reduce method chaining:** Avoid long chains of method calls or property accessors, as they can make your code fragile and tightly coupled.

Clean Code - Design rules 6

Follow Law of Demeter. A class should know only its direct dependencies.

```java
class SparkPlug {
    void fire() {
        System.out.println("Spark plug fired");
    }
}

class Engine {
    private SparkPlug sparkPlug;

    Engine(SparkPlug sparkPlug) {
        this.sparkPlug = sparkPlug;
    }

    void start() {
        sparkPlug.fire();
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;
    private SparkPlug sparkPlug;

    Car(Engine engine, SparkPlug sparkPlug) {
        this.engine = engine;
        this.sparkPlug = sparkPlug;
    }

    void start() {
        // The Car class directly interacts with SparkPlug, which it shouldn't know about.
        // It should only interact with the Engine.
        sparkPlug.fire();
        engine.start();
        System.out.println("Car is ready to go");
    }
}

public class Main {
    public static void main(String[] args) {
        SparkPlug sparkPlug = new SparkPlug();
        Engine engine = new Engine(sparkPlug);
        Car car = new Car(engine, sparkPlug);
        car.start();
    }
}
```

Clean Code - Design rules 6

Follow Law of Demeter. A class should know only its direct dependencies.

```java
class SparkPlug {
    void fire() {
        System.out.println("Spark plug fired");
    }
}

class Engine {
    private SparkPlug sparkPlug;

    Engine(SparkPlug sparkPlug) {
        this.sparkPlug = sparkPlug;
    }

    void start() {
        sparkPlug.fire();
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;

    Car(Engine engine) {
        this.engine = engine;
    }

    void start() {
        engine.start();
        System.out.println("Car is ready to go");
    }
}

public class Main {
    public static void main(String[] args) {
        SparkPlug sparkPlug = new SparkPlug();
        Engine engine = new Engine(sparkPlug);
        Car car = new Car(engine);
        car.start();
    }
}
```

Clean Code -Understandability Rules

- Be consistent, do similar things in similar ways
  - E.g.: Load() & Save(), Open() & Close(), ReadFile() & WriteFile()
- Use explanatory variables over comments

```
13  public double calculateArea() {
14          // Use the formula Area = pi * radius * radius to calculate the area
15          return Math.PI * radius * radius;
16      }
17
18      public double calculateArea() {
19          double area = Math.PI * radius * radius; // Using explanatory variable
20          return area;
21      }
22
```

Clean Code -Understandability Rules

- **Encapsulate boundary conditions, Put the processing for them in one place (don't repeat conditions)**

```java
/**
     * Allows or denies the request based on the rate limit.
     * @return true if the request is allowed, false otherwise.
     */
    public boolean allowRequest() {
        long currentTime = System.currentTimeMillis();
        if (isSameSecond(currentTime, lastRequestTime)) {
            requestCount++;
            // Encapsulated boundary condition
            return isWithinRateLimit(requestCount);
        } else {
            lastRequestTime = currentTime;
            requestCount = 1;
            return true;
        }
    }

    private boolean isSameSecond(long currentTime, long previousTime) {
        return (currentTime / 1000) == (previousTime / 1000);
    }

    // Encapsulates the boundary condition in one place
    private boolean isWithinRateLimit(int count) {
        return count <= MAX_REQUESTS_PER_SECOND;
    }
```

Clean Code -Understandability Rules

- Prefer dedicated value objects to primitive types (e.g. Money not float) (avoid “primitive obsession”)

```java
public class Book {
    private String title;
    private String author;
    private int year; // Primitive type used for year

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

    // Getters and setters...
}
```

Clean Code -Understandability Rules

Prefer dedicated value objects to primitive types

```java
public class Book {
    private String title;
    private String author;
    private Year year; // Dedicated value object used for year

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

    // Getters and setters...
}

public class Year {
    private final int year;

    public Year(int year) {
        if (year < 1450 || year > java.time.Year.now().getValue()) {
            throw new IllegalArgumentException("Year is not valid");
        }
        this.year = year;
    }

    public int getYear() {
        return year;
    }

    @Override
    public String toString() {
        return String.valueOf(year);
    }
}
```

Clean Code -Understandability Rules

Avoid logical dependency, don't write methods that depend on other components of the same class having done something

```java
public class UserProfile {
    private String name;
    private String address;

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public void displayProfile() {
        if (name == null || address == null) {
            System.out.println("Name and address must be set before displaying the profile");
            return;
        }
        System.out.println("Name: " + name + ", Address: " + address);
    }
}
```

Clean Code -Understandability Rules

Avoid logical dependency, don't write methods that depend on other components of the same class having done something

```
1  public class UserProfile {
2      private final String name;
3      private final String address;
4
5      public UserProfile(String name, String address) {
6          this.name = name;
7          this.address = address;
8      }
9
10     public void displayProfile() {
11         System.out.println("Name: " + name + ", Address: " + address);
12     }
13 }
```

Clean Code -Understandability Rules

- Avoid negative conditionals:
    - if (user.isAuthorized()) { giveAccess(); } vs. if (!user.isNotAuthorized()) { giveAccess(); }

Clean Code - Be Explicit, NOT Implicit

- “Ambiguities and imprecision in code are either a result of disagreements or laziness. Either way they should be eliminated.” - Clean Code, Uncle Bob
- Avoid using hidden / automated mechanics of languages, or at least make it obvious when doing so (KISS)

Clean Code - Naming Rules

- Choose descriptive and unambiguous names.
- Make meaningful distinction.
- Use pronounceable names.
- Use searchable names.
- Replace magic numbers with named constants.
- Avoid encodings. Don't append prefixes or type information.

Clean Code - Function Rules

1. Small
    a. This is probably the most important thing I can teach you, everything falls apart and goes bad when functions get long
2. Do one thing. Do it well. Do it only.
3. Use descriptive names
4. Prefer fewer arguments
5. Have no side effects
6. Don’t use control flag arguments (booleans)
    a. Shape s = new Shape(); s.rotate(30, true); // ← What’s that do?
    b. void rotate(int amount, boolean radians); // ← Lame

Clean Code - Comment Rules

- Always try to explain yourself in code.
- Don't be redundant.
- Don't add obvious noise.
- Don't use closing brace comments.
- Don't comment out code. Just remove.
- Use as explanation of intent.
- Use as clarification of code.
- Use as warning of consequences.

Clean Code – Source Code Structure

- Separate concepts vertically.
- Related code should appear vertically dense.
- Declare variables close to their usage.
- Dependent functions should be close.
- Similar functions should be close.
- Place functions in the downward direction.
- Keep lines short.
- Don't use horizontal alignment.
- Use white space to associate related things and disassociate weakly related.
- Don't break indentation.

Clean Code - Formatting

Code formatting is crucial for communication and should be one of the top priorities for professional developers. Proper code formatting and readability have **long-lasting effects on maintainability and extensibility**, even when the original code has changed significantly. - Uncle Bob

Clean Code - Formatting

Using Formatter: **google-java-format Plugin**

To Install :

- Open IntelliJ IDEA.
- Go to **File** > **Settings** (or press **Ctrl+Alt+S** on Windows/Linux, or **Cmd+,** on macOS).
- In the Settings window, select **Plugins** from the left sidebar.
- Click the **Marketplace** tab near the top of the window.
- In the search bar, type **google-java-format** and press **Enter**.
- From the search results, locate the **google-java-format** plugin and click the **Install** button.
- After the installation is complete, click the **Restart IDE** button to restart IntelliJ IDEA and activate the plugin.

Clean Code - Formatting (Cont.)

To Enable and Use:

- Enable plugin for the current project: **Ctrl+Alt+S** → Other Settings → google-java-format Settings and then check the **Enable google-java-format** checkbox.
- Configure IntelliJ JRE: Go to Help → Edit Custom VM Options and paste the provided lines of code.

```
--add-exports=jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED
--add-exports=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
--add-exports=jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED
--add-exports=jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED
--add-exports=jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED
--add-exports=jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED
```

- Restart the IDE after configuring VM options.
- Then you can use hotkey like **Ctrl+Alt+L** or you can define your own hot key for it

Clean Code - Formatting (Cont.)

Ignore formatting comments.

```
/**
 * Processor contains the processing logic for responding
 * to somebody's profile change events from example system One.
 *
 * Change Log
 * Ticket/Date      User        Description
 *
 ----------------------------------------------------------------------

 * FP#136999        shuntian    - added missing check for a  ID
 *                              - added missing response message.
 *                              - added missing email notification.
 */
```

Clean Code - Formatting (Cont.)

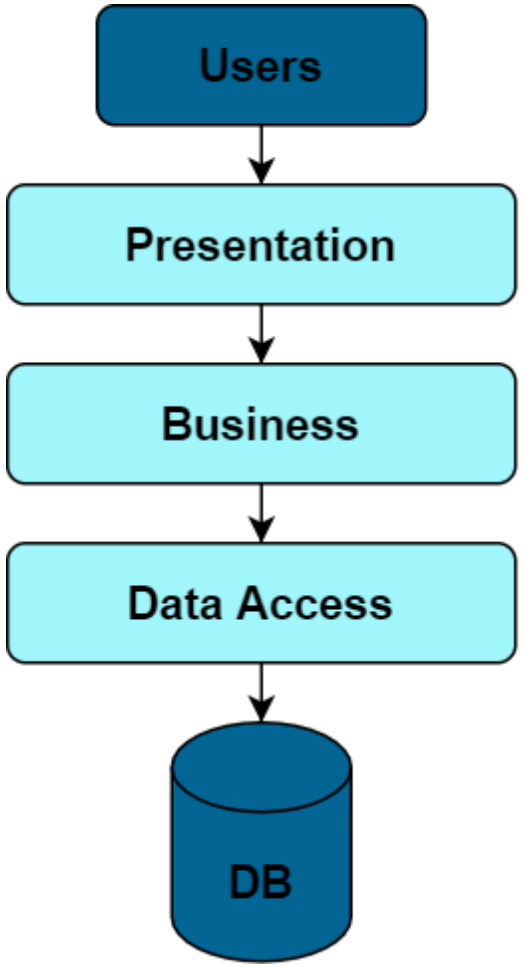
Dis- formatted after format.

```
/**
 * Processor contains the processing logic for responding to somebody's profile
 change events from
 * example system One.
 *
 * <p>Change Log Ticket/Date User Description
 *
 ----------------------------------------------------------------------------
 *
 * <p>FP#136999 shuntian - added missing check for a ID - added missing response
 message. - added
 * missing email notification.
 */
```

MCV and Layers

Layers

- logical separation of concerns within an application.



Persistence Layer / Data Access Layer

- Provides persistent storage of some kind (RDBMS, NoSQL, file system, etc.)
  - Saving data
  - Loading data
  - Alteration of data for storage (serialization, compression, etc…)
- What goes in this layer?
  - Data, the more raw the better
  - Data separated into its smallest elements
  - Ideally in third normal form
  - Data transformation (converting from in memory model to storage format and vice versa)
- What does not go in this layer?
  - "Thinking" - application logic
  - Display information (data wrapped in display info)
  - Rule information stored with data

Presentation / Display Layer

- Displaying data to the user, receiving input, passing input on to the application
- What logic goes in this layer?
  - Display logic
  - Formatting logic
  - User input validation (with a caveat, try to use the business layer if possible, this may come in handy if you decide to port that logic to something else down the road)
- What logic does not go in this layer?
  - Deciding what data to display based on contents of the data
  - Saving / loading data in the data layer
  - Altering data
- Goal:
  - Try to have your presentation layer extremely simple / dumb
  - By keeping the layer dumb, you can swap it for any technology

Business Logic / Rules Layer

- Part of the program that encodes business rules that determine data creation and alteration, the connection between data display and data persistence
  - How business objects interact with each other
  - Rules about data (how it can be altered, valid values, valid operations)
  - How data is organized (ownership, hierarchy, structural and behavioural patterns)
- What goes in this layer?
  - Anything not concerned with display or persistence of data
  - Logic related to the reason your application even exists (what problem does your application solve? The solution is in this layer)
- What does not go in this layer?
  - Decisions about HOW to store data (compressed, cached, storage technology, etc…)
  - Decisions about WHERE to store data
  - Altering data to affect how it is displayed (adding display logic to it)

Is This A Good Idea? (Presentation Layer)

```
<html>
    <body>
    <h1>Student Record</h1>
    <div>ID: <%= student.getId()%></div>
    <div>First Name: <%= student.getFirstName()%></div>
    <div>Last Name: <%= student.getLastName()%></div>
    <div>Is Foreign Student: <%= student.getCountry() != "Canada"></div>
    </body>
</html>
```

Is This A Good Idea? (Business Layer)

```
Sub UpdateCEUsersSaveInfo(strCEUserName)
  Dim rs, sql, bSaveAddress
  sql = "SELECT * " &
        "FROM CEUsers C " &
        "INNER JOIN CEGroups CG ON C.SubscriberID = CG.SubscriberID " &
        "WHERE C.CEUserName = '" & Replace(strCEUserName, "'", "''") & "'"
  Set rs = objConn.Execute(sql)
  If Not rs.EOF Then
    bSaveAddress = True
    sql = "UPDATE CEUsers SET SaveName = 1"
    Do While Not rs.EOF
      If rs("MembersMustPay") Then
        bSaveAddress = False
        Exit Do
      End If
      rs.MoveNext
    Loop
    If bSaveAddress Then
      sql = sql & ", SaveAddress = 1"
    End If
    sql = sql & " WHERE CEUserName = '" & Replace(strCEUserName, "'", "''") & "'"
    Set rs = Nothing
    objConn.Execute(sql)
  End Sub
```

Is This A Good Idea? (Data Layer)

```
CREATE TABLE University (
  Name VARCHAR(MAX),
  DisplayHeader VARCHAR(MAX),
  Address VARCHAR(MAX),
  ContactInfo VARCHAR(MAX)
)
```

```
<html>
  <h1><%= university.GetName() %></h1>
  <%= university.GetDisplayHeader() %>
  <h2>Address:</h2>
  <%= university.GetAddress() %>
  <h2>Contact Info:</h2>
  <%= university.GetContactInfo() %>
</html>
```

```
INSERT INTO University (Name, DisplayHeader, Address, ContactInfo)
VALUES (
  'Dalhousie',
  '<b>Dalhousie is great!</b>',
  '<br/><h3>1 Street</h3><br/><h3>Halifax, Nova Scotia</h3>',
  '<br/><h3>info@dal.ca</h3><br/>'
);
```

Why Not?

- Inflexible, cannot swap solutions in layers
- Maybe it works when you first write it, but what about when something changes?
  - You can't safely make changes in a layer without worrying about breaking other layers! They are disconnected and not necessarily proximal, so this is not easy.
  - **Rigidity is a disease, it spreads and kills your business**
    - A major customer is considering renewing their subscription to your company's products, they don't like something about your application, but rigidity prevents you from changing it. You lose that customer.
    - A competitor implements something that changes the game, if you don't follow suit you go out of business.
    - A new law or regulation comes into effect, you must meet the regulation or suffer massive fines or be shut down
- Let's look at some examples

Competition - Failure / Inability To Innovate

- Adapt or Die:
  - Like evolution, fundamental rule in business. You cannot stagnate, you must be at the front of the pack, not the back
- Blockbuster:
  - Should have been Netflix
  - Would it have been easy to go digital? Imagine the struggle trying to scale.
    - Netflix started by mailing DVDs around
    - Cable companies started streaming / on-demand
    - Too little too late from Blockbuster
- Sears:
  - Should have been Amazon
- RIM (Blackberry):
  - Should have been Apple / iPhone
  - Even today they still aren't caught up? Why? Probably rigidity.

Solution? Separation of Concerns

- Achieve Cohesion
  - Bundle modules into their concerns using the cohesion principles
    - Presentation / Display
    - Business Logic / Rules
    - Data Layer / Persistent Storage
- Avoid Coupling
  - You will be faced with a choice: get your program working quickly vs. preventing coupling and keeping your modules free from rigidity so that you can swap modules in and out
    - Choose the difficult path, what you're writing either has value or it doesn't, if it has no value why are you writing it? Go find an existing solution.
    - If it has value, write it for the long term to maximize re-use and flexibility. Otherwise don't write it!
- Most common solution: Three-Tier Architecture (presentation, business, data)

Model - View - Controller (MVC)

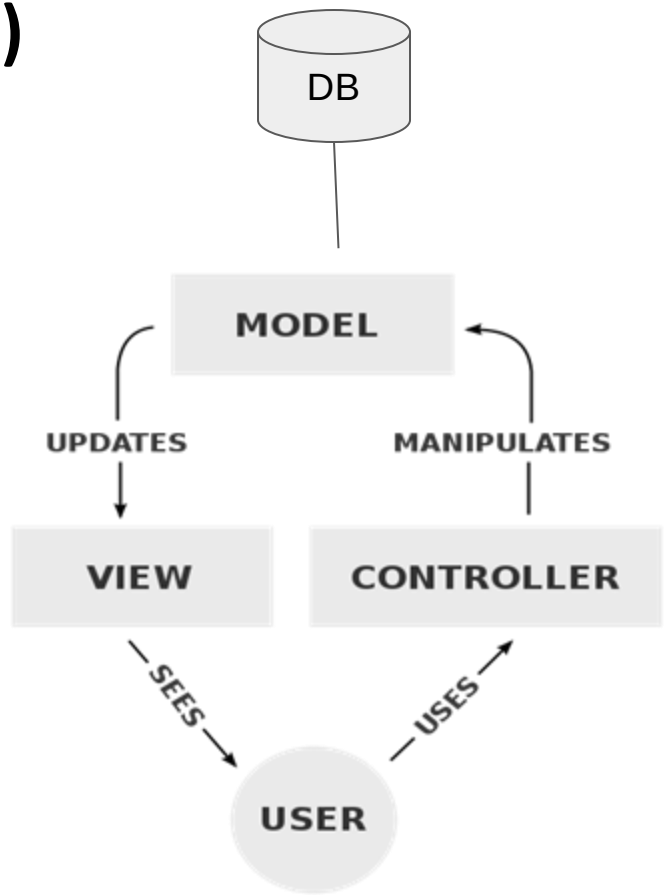
MVC is a design pattern that divides an application into three interconnected components:

- **Model**: The Model is the component of the application that handles the logic for the application data. Often, model objects retrieve data (and store data) from a database.
- **View**: A View is that part of the application that handles the display of the data. Most often the data are shown in a table, but there are many other ways to view the same, or altered, data.
- **Controller**: The Controller is the component that handles user interaction. Typically, the user interacts with the View, which then raises events that are handled by the Controller. The Controller interacts with the Model, possibly updating it in response to user actions.

Model - View - Controller (MVC)

Language examples:

- Java: Struts, Spring mvc, etc…
- Python: Legos
- ASP.Net MVC
- These systems give you the structure to start from and help you maintain cohesion and put your code in the right place



MVC Solution To Earlier Examples (Display Layer)

```
<html>
  <body>
    <h1>Student Record</h1>
    <!-- Notice how everything here is dumb and simple getters -->
    <div>ID: <%= student.getID()%></div>
    <div>First Name: <%= student.getID()%></div>
    <div>Last Name: <%= student.getID()%></div>
    <div>Student Type: <%= student.getIsForeignStudent()%></div>
  </body>
</html>

public class student {
  private String country;
  public String getIsForeignStudent() {
    if (country.equals("Canada")) { return "Resident"; }
    else {return "Foreign"; }
  }
}
```

MVC Solution To Earlier Examples (Business Layer)

```
Sub UpdateCEUsersSaveInfo(strCEUserName)
  Dim user = CEDataLayer.LoadUser(strCEUserName)
  user.SaveName = True
  If Not user.MembersMustPay Then
      user.SaveAddress = True
  End If
  CEDataLayer.UpdateUser(user)
End Sub
```

MVC Solution To Earlier Examples (Data Layer)

```
CREATE TABLE University(
  ID BIGINT,
  Name VARCHAR(MAX),
  Slogan VARCHAR(MAX)
)
CREATE TABLE ContactInfo(
  ID BIGINT,
  Street VARCHAR(MAX),
  City VARCHAR(MAX),
  Province VARCHAR(MAX),
  Email VARCHAR(MAX)
)
```

```
<html>
  <h1><%= university.GetName() %></h1>
  <br/><b><%= university.GetSlogan() %></b>
  <h2>Address:</h2>
  <br/><h3><%= university.GetStreet() %></h3>
  <br/>
  <h3>
    <%= university.GetCity()%>,
    <%= university.GetProvince()%></h3>
  <h2>Contact Info:</h2>
  <br/><h3><%= university.GetEmail() %></h3>
</html>
```

```
INSERT INTO University (Name, Slogan)
VALUES ('Dalhousie', 'Dalhousie is great!');

INSERT INTO ContactInfo (ID, Street, City, Province, Email)
VALUES ('1 Street', 'Halifax', 'Nova Scotia', 'info@dal.ca');
```

Where Should Heavy Duty Data Processing Go?

- Business logic in the DB: Good or Bad?
  - **Bad:**
    - Locks you into a specific RDBMS
    - Splits logic into multiple places
    - Harder to debug database logic
  - **Good:**
    - RDBMS are powerful tools capable of insanely powerful operations revolving around sets of data
    - Potentially more scalable (spin up more servers)
- **Performance**: Consider this last, focus on maintainable / cohesive code. Optimize when performance is for sure a problem
- **Understandability**: SQL is not the simplest language in the world, people do not think in terms of sets
- **Testability**: Possible, yet unlikely you will use test-driven development in SQL

Rob's Advice - DB / Business Layer

- Start with **no** logic in the DB:
  - **Only exception is sorting**, if you are sure your data will be sorted you should default it to its most common sorting arrangement on return from the DB.
- ZERO raw SQL in your business layer:
  - Use stored procedures / functions to load and save objects
  - Protects you from schema changes in your database
- When and if you have performance issues:
  - Use profiling tools to ensure that processing large volumes of data in memory is for sure the problem and not something else. Assumptions are the curse of our industry, be sure.
  - Optimize in stages. Find the largest set of data that gets worked on, move that to a stored procedure or function, re-analyze your performance, repeat.