Package ec.edu.espe.EDICOMPUCMS.controller			
Class			
Long Methods	public static void backupComputerStatuses(String fileName) {...}	The `backupComputerStatuses` method performs multiple tasks: generating and writing statuses, which can make it hard to read.	Break the method into several smaller methods with clear responsibilities. For example, `generateStatuses` and `writeStatusesToFile`
Conditional Complexity	`catch (IOException e) {...}`	Exception handling in the `writeStatusesToFile` method may become more complex if additional conditions are added.	Consider creating a separate method for exception handling or simplifying the error handling code.
Class CustomerManager			

Long Methods	`public void addCustomer(Customer customer) {...}`
	`public void addUser(Users user) {...}`
	`public Users getUser(String username) {...}`
	`public List<Customer> getCustomers() {...}`
	`public void updateCustomer(Customer customer) {...}`
	`public void removeCustomer(String id) {...}`
		Methods such as `addCustomer`, `addUser`, `getUser`, `getCustomers`, `updateCustomer`, and `removeCustomer` have multiple lines of code and responsibilities, which can make them hard to read.	Break down the methods into smaller methods with clear responsibilities. For example, extract document conversion logic into auxiliary methods.

Large Classes	The `CustomerManager` class contains several methods related to managing customers and users.	The `CustomerManager` class has multiple responsibilities, which can make it difficult to maintain and understand.	Refactor the class into multiple classes with separate responsibilities, such as `CustomerService` and `UserService`.

Inappropriate Intimacy	`public CustomerManager() { ... }`
	`MongoDatabase database = DatabaseConnection.getInstance().getDatabase();`
		The `CustomerManager` class is tightly coupled with `DatabaseConnection`, which can make it difficult to change or test.	Use dependency injection to pass `MongoCollection` and `DatabaseConnection` instead of creating instances within the class.

Class CyberManager			
Large Classes	public class CyberManager {...}	The `CyberManager` class has multiple responsibilities, including managing computers, tariffs, history, and listeners.	Refactor into multiple classes, such as `ComputerManager`, `TariffManager`, and `HistoryManager`
Long Methods	public double stopComputer(int id) {...}	The `stopComputer` method performs several tasks, making it harder to maintain.	Split into smaller methods like `calculateCost`, `updateHistory`, and `notifyListeners`.
Class MainMenu			
High Coupling	createCyberManagementPanel	The `MainMenu` class is directly creating instances of specific panels (`CustomerManagementPanel`, `CyberManagementPanel`, etc.),	Consider dependency injection or using a panel factory to reduce coupling.

		increasing coupling between the UI and business logic. This can make the application harder to test and maintain.	
Single Responsibility	MainMenu	The `MainMenu` class handles multiple responsibilities: UI management, login/logout logic, and clock updates.	Consider refactoring to delegate responsibilities to more specialized classes.
Literal Usage	`"Home", "Customers", "CyberManagement", "Payments"` in `createMainPanel`	String literals are used directly to identify the cards in the `CardLayout`, which can lead to hard-to-trace errors if there are typos.	Define constants for the card names and use them instead of literals.
Class RentalManager			
Single Responsibility	RentalManager	The class manages customers, computers, and rentals, leading to too much responsibility.	Consider dividing logic into more specific classes.
Code Duplication	`findCustomerById`, `findComputerById`	Search methods duplicate logic.	Consolidate into a generic method or use `Stream`.
Literal Usage	System.out.println	Hardcoded messages are used directly in the code.	Use constants or externalize messages for easier updates.
Class			
Lack of Flexibility	TariffManager	The tariff is only initialized upon creating the class, and is not modifiable afterward.	Add methods to update tariffs if needed.
Duplication of Responsibility	TariffManager	The class handles both creating and accessing the tariff, but could be simplified.	Consider separating tariff creation into another class or method.

Package ec.edu.espe.EDICOMPUCMS.model			
Class Computer			
Long Methods	getActiveDuration()	The method handles multiple conditions and calculations, which makes it lengthy and harder to read.	Split getActiveDuration() into smaller methods to handle each condition separately, improving readability.
Duplicate Code	getActiveDuration()	Code for calculating duration is duplicated for the active and inactive states.	Refactor to use a common method for calculating duration based on conditions.
Feature Envy	calculateCost()	The method depends on Tariff for cost calculation, indicating that Tariff should manage cost-related logic.	Move calculateCost() to the Tariff class if the cost calculation is more closely related to the tariff's role.

Inappropriate Intimacy	start(), stop(), getActiveDuration()	These methods manipulate internal states directly, showing excessive knowledge of internal workings.	Consider encapsulating internal state management and providing higher-level methods to interact with these states.
Speculative Generality	setActive(boolean active)	This method may be unnecessary if there is no other use case for setting the active state outside of start() and stop().	Remove setActive() if it's redundant or not used elsewhere; ensure the class has only necessary methods.
Class Customer			
Large Classes	Whole class	The class handles multiple attributes and methods, making it larger than necessary.	Consider splitting into smaller classes if it grows or handles more complex behavior.
Speculative Generality	Customer class	The class has a lot of attributes and setters, which may not be needed for all use cases.	Remove or consolidate attributes and methods if they are not required.
Uncommunicative Names	getId(), setId()	Names are generic and do not convey specific purpose or context.	Use more descriptive names if additional context is needed.
Class CustomerMenu			
Long Methods	customerMenu() method	The method has multiple responsibilities and long blocks of code.	Break down into smaller methods for better readability.
Conditional Complexity	customerMenu() method	Nested conditionals in ID and phone validation make the logic complex.	Simplify validation logic or use helper methods.
Redundant or Meaningless Comments	Commented out showCustomers() method	Commented code that should be either removed or implemented.	Remove or complete the commented code.
Feature Envy	CustomerMenu class	The class is heavily dependent on CustomerManager for core functionalities.	Consider moving some functionalities into CustomerManager.
Class DatabaseConnection			
Long Methods	Constructor and getInstance() method	The constructor performs both connection setup and error handling.	Move connection setup to a separate method.
Speculative Generality	CONNECTION_STRING and DATABASE_NAME fields	These fields are hardcoded and not parameterized.	Use configuration files or environment variables for configuration.
Inappropriate Intimacy	DatabaseConnection class	This class manages both connection and database operations.	Separate connection management from database operations.
Feature Envy	DatabaseConnection class	The class is tightly coupled with MongoDB-specific details.	Abstract database operations to reduce direct dependency on MongoDB.
Class FinancialReportMenu			
Inappropriate Intimacy	FinancialReportMenu class	FinancialReportMenu directly handles user input and manages CyberManager.	Separate user interaction and CyberManager management.
Large Classes	FinancialReportMenu class	The class is responsible for multiple concerns (user input handling, managing CyberManager).	Refactor into smaller classes focusing on a single responsibility.
Feature Envy	handleFinancialReport method	This method is heavily dependent on the CyberManager class.	Move CyberManager related logic into its own dedicated class.

Package ec.edu.espe.EDICOMPUCMS.view			
Class CustomerManagementPanel			
Large Class	CustomerManagementPanel class	The class handles UI, validation, and data management.	Refactor into smaller classes for UI, validation, and data management.
Long Methods	addCustomer, updateCustomer, deleteCustomer, searchCustomers, loadCustomers methods	Methods are lengthy and do multiple tasks.	Break down methods into smaller, more focused methods.
Feature Envy	Methods relying on CustomerManager	The class relies heavily on CustomerManager.	Consider moving some logic into CustomerManager or another dedicated class.
Inappropriate Intimacy	setNumericOnly method	Directly manipulates text field documents.	Abstract document manipulation into a separate utility class.
Class CyberManagementPanel			
Large Class	CyberManagementPanel class	The class handles UI setup, timer management, and business logic.	Refactor into separate classes for UI, timer management, and business logic.
Long Methods	startComputer, stopComputer, updateComputerStatus	Methods are lengthy and handle multiple tasks.	Break down methods into smaller, focused methods.
Duplicate Code	createStyledLabel, createStyledButton	Similar styling logic is repeated for labels and buttons.	Consolidate styling code into reusable utility methods or classes.
Inappropriate Intimacy	startComputer, stopComputer	Directly manipulates cyberManager and UI elements.	Consider moving some logic into CyberManager or a separate service class.
Magic Numbers	Timer interval 1000 milliseconds	The interval for the timer is hard-coded.	Define timer intervals as constants with descriptive names.
Hard-Coded Path	Image path in createComputerPanel	Uses a hard-coded file path for images.	Use a relative path or resource loading method to handle image files.