CS 6356: Software Maintenance, Evolution & Re-engineering

Assignment 3: Coupling and Cohesion

TEAM 13

Team Members:

1. Gaurav Sharma (GXS230001) – Contribution: 35%
2. Shalin Ronakkumar Kaji (SXK220263) – Contribution: 35%
3. Aditya Krishna (AXK230001) – Contribution: 30%

Contribution:

1. Gaurav Sharma completed the code analysis for both the systems for Cohesion metrics.
2. Shalin Kaji completed the code analysis for both the systems for Coupling metrics.
3. Aditya Krishna compiled the report and generated XML & CSV metric files.

Location of uploaded XML & CSV files for Cohesion and Coupling metrics on GitHub:
https://github.com/gauravsharma2/JeditTeam13/tree/18979821b6e00667d6b3604ee1104305d8e7367c/Metrices%20Files

file/folder of the metrics are present in the jEdit repository inside JeditTeam13/Metrices Files/ in the repositories.

We used the following tools and plugins in the IntelliJ IDE for evaluating various Cohesion and Coupling metrics in classes of two systems: **MangoDB** and **jEdit**.

1. Metrics Reloaded for IntelliJ
2. Inspect Code – IntelliJ
3. CodeMR Plugin – IntelliJ

Description of the metrics selected: -

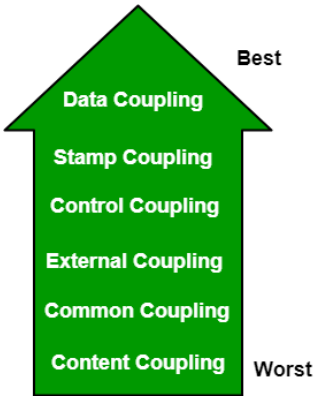
Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design.

1. Coupling:

Coupling refers to the degree of interdependence between software modules.

High coupling means that modules are closely connected and changes in one module may affect other modules.

Low coupling means that modules are independent and changes in one module have little impact on other modules.



Types of Coupling:

Data Coupling: The modules are considered data coupled if their dependence is predicated on the fact that they exchange solely data for communication. The components of a data coupling are separate from one another and communicate via data. Tramp data is absent from module communications. Customer billing system, for instance.

Stamp Coupling The entire data structure is transmitted from one module to another in a stamp coupling. Thus, tramp data is involved. Efficiency considerations might make it necessary; the intelligent designer, not a slothful coder, made this decision.

Control Coupling: The modules are referred to as control linked if they exchange control information with one another. If arguments allow for factoring and functionality reuse, that can be good; however, it can be negative if the parameters show entirely different behavior. A sort function that accepts the comparison function as an argument is an example.

External Coupling: When modules are coupled externally, they rely on one another rather than the software or a certain kind of hardware that is being produced. Device format, external file, protocol, etc.

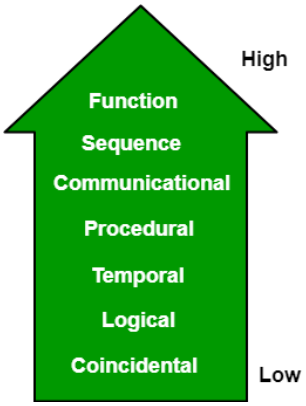
Common Coupling: Global data structures and other shared data are present in the modules. To assess the impact of a modification to global data, all modules that access it must be tracked back to. Thus, it has drawbacks such as limited maintainability, less control over data access, and difficulties reusing modules.

Content Coupling: One module can change the data of another module or transfer control flow from one module to the other in a content coupling. The worst kind of coupling is this one, therefore stay away from it.

2. Cohesion:

Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a

single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Types of Cohesion:

Functional Cohesion: Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.

Sequential Cohesion: An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
Communicational Cohesion: Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.

Procedural Cohesion: Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

Temporal Cohesion: The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.

Logical Cohesion: The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.

Coincidental Cohesion: The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst

form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Advantages of low coupling:

- Enhanced maintainability: Modifying or replacing individual parts without affecting the system as a whole is made easier by low coupling, which lessens the effect of changes in one module on other modules.
- Enhanced modularity: Code may be produced and tested independently thanks to reduced coupling, which increases code's modularity and reusability.
- Better scalability: The system can be scaled more easily as needed since little coupling makes it easy to add new modules and remove old ones.

Advantages of high cohesion:

- Better readability and understandability: Clear, focused modules with a single, well-defined goal are the outcome of high cohesiveness, which facilitates developer comprehension and modification of the code.
- Improved error isolation: Strong cohesiveness lessens the possibility that modifications to one module component will have an impact on other components, which facilitates
- Enhanced dependability: Strong cohesiveness results in modules that operate more consistently and with less error-proneness, enhancing the system's overall dependability.

Disadvantages of high coupling:

- Enhanced complexity: A system with a high coupling level has more interdependencies between its parts, which makes it more complex and challenging to comprehend.
- Less flexibility: It is harder to replace or alter specific parts of a high coupling system without also altering the system as a whole.
- Reduced modularity: Code that has a high coupling level is less modular and reusable since it is more difficult to build and test parts separately.

Disadvantages of low cohesion:

- Greater code duplication: When components that should work together are divided into different modules, a lack of cohesiveness can result in an increase in code duplication.
- Reduced functionality: Modules with low cohesiveness may have pieces that aren't meant to be together and serve no obvious purpose, which limits their usefulness and makes maintenance more difficult.
- Difficulty comprehending the module: A module with low cohesiveness may be more difficult for developers to comprehend, which can result in mistakes and unclear behavior.

Measuring and Interpreting Coupling

Coupling refers to the degree of interconnectedness between modules or classes within a system. High coupling can make code harder to maintain, understand, and modify, while low coupling generally leads to more modular, flexible, and maintainable code.

These metrics are selected for studying Coupling in the codebase:

1. **Coupling Between Objects (CBO):**
    - CBO measures the number of other classes a class is coupled to.
    - It indicates how many other classes a particular class is directly dependent on or interacts with.
    - High CBO values suggest tight coupling, where changes in one class may require changes in many other classes.
    - Low CBO values indicate loose coupling, meaning classes are more independent and changes in one class are less likely to impact others.
2. **Response For a Class (RFC):**
    - RFC measures the number of different methods that can be executed in response to a message received by an object of a class.
    - It represents the potential behavior encapsulated within a class.
    - Higher RFC values indicate that the class has more responsibility and may perform a wide range of actions in response to messages.
    - Lower RFC values suggest a more focused and cohesive class with a narrower range of responsibilities.

In terms of coupling metrics:

- CBO is more focused on the relationships between classes, indicating how tightly they are coupled.
- RFC is more focused on the internal complexity and responsibilities of a single class.

jEdit Codebase:

Now based on the metric of CBO, we would like to evaluate the jEdit Codebase:

1. **Three non-trivial classes with the tightest (highest = high CBO value) coupling:**

- *org.gjt.sp.jedit.jEdit.java*
- *org.gjt.sp.jedit.View.java*
- *org.gjt.sp.util.Log.java*

Explanation:

View.java:

- **Content Coupling:** There is content coupling observed in **View.java** with **DockableWindowManager**. The **View** class relies on methods and constants from **DockableWindowManager** such as **getDockableWindowManager()** and various layer constants. For example, methods like **addToolBar()** and **removeToolBar()** in **View.java** interact directly with the **DockableWindowManager**.
- **Control Coupling:** Control coupling is evident in methods like **processKeyEvent()**, where the **View** class controls the flow of events by passing key events to **InputHandler** for processing.

Log.java:

- **Content Coupling:** Content coupling is observed in **Log.java** with the use of **System.out** and **System.err**. The **init()** method redirects standard output and error streams to custom print streams created within **Log.java**. This reliance on system-level output streams ties the behavior of **Log.java** to the behavior of the system's standard output and error handling.
- **Control Coupling:** Control coupling is evident in the **log()** method, where the urgency level of the log message determines whether the message is printed to standard output (**System.out**) or standard error (**System.err**). This control flow decision is based on the urgency parameter passed to the method.

jEdit.java:

- **Content Coupling:**

  jEdit class has content coupling with various other classes such as **View**, **DockableWindowManager**, **ServiceManager**, and **Log**. For instance, methods like **getDockableWindowManager()**, **getLogListModel()**, and **getBeepOnOutput()** are directly called from **jEdit** class, indicating content coupling.

- **Common Coupling:**

  There is common coupling observed in **jEdit** class with the use of constants like **VIEW_DOCKING_FRAMEWORK_PROPERTY** and

DOCKING_FRAMEWORK_PROVIDER_SERVICE. These constants are shared among multiple classes, implying common coupling.

- **Control Coupling:**

  Control coupling is evident in methods like **init()** and **setBeepOnOutput()** where **jEdit** class controls the behavior of other classes by setting up logging or determining if beeping is enabled.

- **Data Coupling:**

  Data coupling is observed in methods like **init()** and **setBeepOnOutput()**, where data such as logging level or beep status is passed as arguments to other classes or methods.

2. **Three non-trivial classes with the most loose (low = low CBO value) coupling:**

- *org.gjt.sp.jedit.ActionContext.java*
- *org.gjt.sp.jedit.JEditBeanShellAction.java*
- *org.gjt.sp.jedit.Abbrevs.java*

Explanation:

ActionContext.java:

- **Single Responsibility Principle (SRP):** The **ActionContext** class appears to manage a collection of action sets, focusing solely on this responsibility. It extends a generic **JEditActionContext** class, indicating a separation of concerns.
- **Abstraction:** The class is abstract, allowing for different implementations to manage action sets. This abstraction enables flexibility and extensibility.
- **Dependency Inversion Principle (DIP):** The class relies on interfaces (**EditAction** and **ActionSet**) rather than concrete implementations, adhering to DIP principles by depending on abstractions rather than concrete classes.

Abbrevs.java:

- **Encapsulation:** The **Abbrevs** class encapsulates abbreviation management functionality, providing methods to interact with abbreviation sets.
- **Limited External Dependencies:** The class depends on standard Java libraries and a few jEdit-specific classes (**View**, **Buffer**, **JEditTextArea**). It does not exhibit excessive dependencies on other components.

- **Clear Purpose:** The purpose of managing abbreviations is clear from the class name and method names, promoting readability and maintainability.

JEditBeanShellAction.java:

- **Encapsulation:** Similar to **Abbrevs**, this class encapsulates functionality related to BeanShell actions, such as invocation and checking if the action is selected.
- **Dependency Injection:** The class uses dependency injection to provide the **BeanShellFacade** instance, allowing for easier testing and flexibility in changing implementations.
- **Clear Separation of Concerns:** The class separates concerns related to BeanShell action execution and state management, improving code organization and readability.

Comparison with Tightly Coupled Classes:

- **Reduced Dependency Graph:** The loosely coupled classes exhibit fewer dependencies on external components compared to the tightly coupled classes. They have clearer boundaries and focus on specific responsibilities, leading to a simpler dependency graph.
- **Abstraction and Interfaces:** Loose coupling is facilitated by the use of interfaces and abstractions, allowing for interchangeable components and reduced reliance on concrete implementations.
- **Clearer Separation of Concerns:** Each class has a clear and focused responsibility, adhering to the Single Responsibility Principle (SRP). This clear separation reduces interdependencies between components and promotes maintainability.
- **Flexibility and Extensibility:** Loose coupling enables easier modification and extension of functionality without impacting other parts of the system. This flexibility allows for smoother evolution of the codebase over time.

Now based on the metric of RFC, we would like to evaluate the jEdit Codebase:

1. **Three non-trivial classes with the tightest (highest = high RFC value) coupling:**
    - *org.gjt.sp.jedit.jEdit.java*

- *org.gjt.sp.jedit.textarea.TextArea.java*
- *org.gjt.sp.jedit.bsh.Parser.java*

Explanation:

jEdit.java:

1. **Multiple Methods Invoked:**

- The **jEdit** class contains several methods (**getVersion()**, **getBuild()**, **main()**, etc.) that can respond to different messages or events. Each of these methods potentially contributes to the RFC value of the class.

2. **External Method Calls:**

- The **main()** method of the **jEdit** class invokes methods from various external classes, such as **MiscUtilities**, **OperatingSystem**, **GUIUtilities**, etc. These method invocations contribute to the RFC value of the **jEdit** class because they represent potential responses to messages received by **jEdit**.

TextArea.java:

1. **Interaction with GUI Components:**

- The **TextArea** class interacts with various GUI components such as **JPanel**, **JScrollBar**, **JLayer**, **BoxLayout**, etc., through method calls like **setLayout()**, **add()**, and **putClientProperty()**. These interactions increase coupling with Swing components and potentially with their associated event listeners and handlers.

2. **Interaction with Buffer and Display Manager:**

- Methods like setBuffer(), getBuffer(), and getDisplayManager() interact with the JEditBuffer and DisplayManager classes, indicating tight coupling with text buffer management and display rendering functionalities.

3. **Dependency on Utility Classes:**

- The TextArea class depends on utility classes like TextUtilities, StandardUtilities, ThreadUtilities, etc., for various functionalities such as text manipulation, standard utility operations, and thread management, leading to increased coupling with these utility classes.

Parser.java:

1. **Integration with Token Parsing Mechanisms:**

- The **Parser.java** class interacts with token parsing mechanisms through methods like **jj_consume_token()** and **getToken()**. These methods are essential for parsing tokens and navigating the input stream, but they tightly couple the parser with the tokenization process.

2. **Parsing Java Language Grammar:**

- The class contains methods for parsing different elements of the Java language grammar, such as class declarations (**ClassDeclaration()**), method declarations (**MethodDeclaration()**), package declarations (**PackageDeclaration()**), and import declarations (**ImportDeclaration()**). These methods directly handle the syntactical rules of the Java language, leading to tight coupling with the language grammar.

2. Three non-trivial classes with loose (lowest = low RFC value) coupling:

- *org.gjt.sp.jedit.textarea.RangeMap.java*
- *org.gjt.sp.jedit.ActionListHandler.java*
- *org.jedit.keymap.KeymapImpl.java*

Explanation:

RangeMap.java:

1. **Limited Interaction with External Classes**: **RangeMap** primarily deals with managing ranges of integers and does not heavily interact with external classes. Its methods mainly manipulate its internal state (**fvm** array) and perform calculations based on integer values.

2. **Encapsulation**: Methods like **reset**, **first**, **last**, **lookup**, **search**, **put**, and **next** operate solely on the **fvm** array or local variables. They do not rely on external dependencies or access external resources.

3. **Limited Method Interactions**: The methods in **RangeMap** mostly perform simple operations such as array manipulation, arithmetic calculations, and comparisons. They do not call a large number of other methods from external classes.

ActionListHandler.java:

1. **Encapsulation of XML Parsing Logic**: **ActionListHandler** encapsulates the logic for parsing XML data related to action lists. It utilizes the **DefaultHandler** class from

org.xml.sax.helpers, but its interaction with this class is limited to overriding methods for handling XML elements and attributes.

2. **Minimal External Method Calls:** Methods like resolveEntity, characters, startElement, endElement, and startDocument are overridden from the DefaultHandler class to handle XML parsing. These methods operate within the context of the XML parsing logic and do not call numerous methods from external classes.

3. **Self-contained Functionality:** ActionListHandler maintains its state using local variables and does not rely heavily on external dependencies. It encapsulates XML parsing functionality within itself, leading to loose coupling with other parts of the codebase.

KeymapImpl.java:

1. **Internal Management of Keymap Properties**: **KeymapImpl** manages keymap properties internally using a **Properties** object. It loads and saves keymap properties to and from a file but does not rely heavily on external classes for its core functionality.

2. **Limited Interaction with External Resources**: Methods like **loadProperties**, **getShortcut**, **setShortcut**, and **save** primarily operate on the **props** object and interact minimally with external resources such as files. They encapsulate keymap management logic within the class.

3. **Self-contained Keymap Functionality**: **KeymapImpl** encapsulates keymap management logic within itself, making it self-sufficient and independent of external dependencies for its core functionality.

Comparison with Tightly Coupled Classes

1. **Dependency on External Classes**: The tightly coupled classes from the previous examples often rely heavily on external classes and libraries for their functionality. They make frequent method calls to external classes, leading to tight coupling and high RFC values.

2. **Complex Interactions**: Tightly coupled classes may have complex interactions with multiple external classes, making them more tightly integrated with the overall codebase. This complexity increases the RFC value due to numerous method calls and dependencies.

3. **Difficulty in Isolation**: Tightly coupled classes are often harder to isolate and test in isolation due to their dependencies on external classes. In contrast, the loosely coupled classes provided are more self-contained and can be tested independently with minimal dependencies.

MangoDB Codebase:

Now based on the metric of CBO, we would like to evaluate the MangoDB Codebase:

1. **Three non-trivial classes with the tightest (highest = high CBO value) coupling:**

- *com.serotonin.mango.Common.java*
- *com.serotonin.mango.vo.DataPointVO.java*
- *com.serotonin.mango.rt.dataImage.PointValueTime.java*

Explanation:

Common.java:

1. **High Coupling:**

- **Common** class has high coupling due to its interaction with the following classes or concepts:
- **RealTimeTimer**: The class includes a static final instance of **RealTimeTimer**, which suggests a tight coupling with this class.
- **MonitoredValues**: Similar to **RealTimeTimer**, **Common** class has a static final instance of **MonitoredValues**, indicating tight coupling.
- **ExportCodes**: **Common** class references **ExportCodes**, implying a dependency on this class.
- **BackgroundContext**: Methods like **getUser()** interact with **BackgroundContext**, indicating coupling with this class.

2. **Type of Coupling Observed:**

- **Content Coupling**: This type of coupling occurs when one class references another class's internal elements, such as its fields or methods. In **Common**, content coupling is observed through direct references to constants (**SESSION_USER**, **UTF8**, etc.) and methods (**getUser()**, **setUser()**) from other classes. For instance, methods like **getUser()** and **setUser()** directly interact with **HttpServletRequest** and **User** objects.

3. **Contrast with Other Types of Coupling:**

- **Common Coupling**: Common coupling would occur if **Common** class and the other classes were using a shared global variable or common environment, without direct references to each other's internal elements. However, in this case, **Common** directly references fields and methods from other classes, indicating content coupling rather than common coupling.
- **Control Coupling**: Control coupling would involve one class controlling the flow of another class through method parameters or return values. There's no evident control coupling in **Common** as it mainly deals with constants and utility methods rather than controlling the flow of execution in other classes.

4. **Examples of Coupling:**

- **getUser()** and **setUser(HttpServletRequest request, User user)**: These methods couple **Common** with the **User** class and **HttpServletRequest** class.
- **getMillis(int periodType, int periods)**: This method couples **Common** with the **Period** class.
- Constants like **SESSION_USER**, **UTF8**: These constants couple **Common** with classes that might use these values.

DataPointVO.java:

1. **High Coupling:**

- **DataPointVO** class exhibits high coupling due to its interaction with the following classes or concepts:
- **Common**: The class references **Common** for constants related to time periods and engineering units, indicating a dependency on this class.
- **ExportCodes**: **DataPointVO** utilizes **ExportCodes** for mapping integer codes to string values, suggesting a dependency on this class.
- **PointEventDetectorVO**: The class contains a list of **PointEventDetectorVO** instances, indicating a relationship with this class.
- **UserComment**: **DataPointVO** includes a list of **UserComment** instances, implying coupling with this class.

2. **Type of Coupling Observed:**

- **Content Coupling**: This type of coupling occurs when one class references another class's internal elements, such as its fields or methods. In **DataPointVO**, content coupling is observed through direct references to constants (**ENGINEERING_UNITS_DEFAULT**, **LOGGING_TYPE_CODES**, etc.), class instances (**PointLocatorVO**), and lists of other objects (**eventDetectors**, **comments**).

- **Control Coupling**: There is no evident control coupling observed in **DataPointVO**. Control coupling would involve one class controlling the flow of another class through method parameters or return values, which is not apparent in this class.

3. **Contrast with Other Types of Coupling:**

- **Common Coupling**: Common coupling would occur if **DataPointVO** and the other classes were using a shared global variable or common environment, without direct references to each other's internal elements. However, in this case, **DataPointVO** directly references fields, constants, and instances from other classes, indicating content coupling rather than common coupling.

- **Control Coupling**: As mentioned earlier, control coupling involves one class controlling the flow of another class through method parameters or return values. Since there's no such interaction observed in **DataPointVO**, it can be concluded that control coupling is not present.

4. **Examples of Coupling:**

- **ENGINEERING_UNITS_DEFAULT**: This constant couples **DataPointVO** with the **Common** class, as it directly references a constant defined in **Common**.

- **LOGGING_TYPE_CODES**: This instance of **ExportCodes** couples **DataPointVO** with the **ExportCodes** class.

- **PointEventDetectorVO** and **UserComment**: The presence of lists containing instances of these classes indicates coupling between **DataPointVO** and **PointEventDetectorVO** and **UserComment**.

PointValueTime.java:

1. **High Coupling:**

- **PointValueTime** class exhibits high coupling due to its interaction with the following classes or concepts:

- **MangoValue**: The class tightly couples with **MangoValue** since it holds an instance of it (**value** field) and relies on its methods (**getDoubleValue()**, **getStringValue()**, etc.) to access value data.
- **DateFunctions**: The class references **DateFunctions** to convert the time value to a human-readable format in the **toString()** method.
- **ObjectUtils**: The class utilizes **ObjectUtils** for value equality comparison (**isEqual()** method).

2. **Type of Coupling Observed:**

- **Content Coupling**: This type of coupling occurs when one class references another class's internal elements, such as its fields or methods. In **PointValueTime**, content coupling is observed through direct references to the **MangoValue** class's methods (**getDoubleValue()**, **getStringValue()**, etc.) and the **DateFunctions** class's **getTime()** method. Additionally, the class uses the **isEqual()** method from **ObjectUtils** to compare values.
- **Contrast with Other Types of Coupling**: It's not control coupling because **PointValueTime** does not control the flow of execution in other classes through method parameters or return values. Also, it's not common coupling because it does not share a common global variable or environment with other classes.

3. **Examples of Coupling:**

- **MangoValue value**: The class tightly couples with **MangoValue** since it holds an instance of it (**value** field).
- **DateFunctions.getTime(time)**: The class couples with **DateFunctions** to format the time value in the **toString()** method.
- **ObjectUtils.isEqual()**: The class couples with **ObjectUtils** to perform value equality comparison in the **equals()** method.

2. Three non-trivial classes with the most loose (lowest = low CBO value) coupling:

- *com.serotonin.mango.db.dao.ViewDao.java*
- *com.serotonin.mango.db.dao.WatchListDao.java*
- *com.serotonin.mango.db.dao.UserDao.java*

Explanation:

ViewDao.java:

1. **Low Coupling:**

- **ViewDao** class exhibits low coupling due to its limited interaction with other classes or concepts:
- The class primarily interacts with the database and relies on JDBC operations (**ejt.update()**, **query()**) for database queries and updates.
- It includes a couple of inner mapper classes (**ViewRowMapper**, **ViewUserRowMapper**) to map result sets to domain objects, but these are tightly scoped and serve only to facilitate data mapping within the class itself.
- There is minimal dependency on external utility classes or domain objects beyond basic JDBC operations.

2. **Type of Coupling Observed:**

- **Low Coupling**: Low coupling indicates that the class is relatively independent and does not heavily rely on other classes or modules. In **ViewDao**, the coupling is primarily focused on database operations, with minimal reliance on external classes or concepts. The class encapsulates its database operations without exposing its internal details to other classes, which contributes to its low coupling.
- **Contrast with Other Types of Coupling**: It's not content coupling because the class does not directly reference internal elements of other classes (apart from inner mapper classes for data mapping, which are tightly scoped within the class itself). Additionally, it's not common coupling as it does not share a common global variable or environment with other classes. Furthermore, it's not control coupling because **ViewDao** does not control the flow of execution in other classes through method parameters or return values.

3. **Examples of Coupling:**

- **ejt.update()**, **query()**: These methods interact with the underlying database, indicating coupling with the database access layer.
- Inner mapper classes (**ViewRowMapper**, **ViewUserRowMapper**): These classes facilitate mapping database result sets to domain objects within the **ViewDao** class, indicating internal coupling for data mapping purposes.

WatchListDao.java:

1. **Low Coupling:**

- **WatchListDao** exhibits low coupling due to its limited interaction with other classes or concepts:
- It primarily interacts with the database through JDBC operations (**ejt.update()**, **query()**) for performing database queries, updates, and batch operations.
- Inner mapper classes (**WatchListRowMapper**, **WatchListUserRowMapper**) are used to map database result sets to domain objects within the **WatchListDao** class itself.
- The class encapsulates its database operations and does not heavily rely on external utility classes or domain objects beyond basic JDBC operations.

2. **Type of Coupling Observed:**

- **Low Coupling**: Low coupling indicates that the class is relatively independent and does not heavily rely on other classes or modules. In **WatchListDao**, coupling primarily revolves around database operations and data mapping, with minimal dependence on external classes or concepts. The class encapsulates its database operations, making it independent of external changes.
- **Contrast with Other Types of Coupling**: It's not content coupling because the class does not directly reference internal elements of other classes. Additionally, it's not common coupling as it does not share a common global variable or environment with other classes. Furthermore, it's not control coupling because **WatchListDao** does not control the flow of execution in other classes through method parameters or return values.

3. **Examples of Coupling:**

- **ejt.update()**, **query()**: These methods interact with the underlying database, indicating coupling with the database access layer.
- Inner mapper classes (**WatchListRowMapper**, **WatchListUserRowMapper**): These classes facilitate mapping database result sets to domain objects within the **WatchListDao** class, indicating internal coupling for data mapping purposes.

UserDao.java:

1. **Low Coupling:**

- It primarily interacts with the database through JDBC operations (**ejt.update()**, **query()**) for performing database queries, updates, and batch operations.
- Inner mapper class (**UserRowMapper**) is used to map database result sets to domain objects within the **UserDao** class itself.
- The class encapsulates its database operations and does not heavily rely on external utility classes or domain objects beyond basic JDBC operations.
- It does not directly interact with other major components or modules in the system, indicating a lack of tight coupling.

2. **Type of Coupling Observed:**

- **Low Coupling**: Low coupling indicates that the class is relatively independent and does not heavily rely on other classes or modules. In **UserDao**, coupling primarily revolves around database operations and data mapping, with minimal dependence on external classes or concepts. The class encapsulates its database operations, making it independent of external changes.
- **Contrast with Other Types of Coupling**: It's not content coupling because the class does not directly reference internal elements of other classes. Additionally, it's not common coupling as it does not share a common global variable or environment with other classes. Furthermore, it's not control coupling because **UserDao** does not control the flow of execution in other classes through method parameters or return values.

3. **Examples of Coupling:**

- **ejt.update()**, **query()**: These methods interact with the underlying database, indicating coupling with the database access layer.
- Inner mapper class (**UserRowMapper**): This class facilitates mapping database result sets to domain objects within the **UserDao** class, indicating internal coupling for data mapping purposes.

Comparison with Tightly Coupled classes:

The comparison between the tightly coupled classes (**Common.java**, **DataPointVO.java**, and **PointValueTime.java**) and the loosely coupled classes (**UserDao.java**, **ViewDao.java**, and **WatchListDao.java**) reveals significant differences in their design and dependencies. The tightly coupled classes exhibit strong interdependencies with other components, resulting in high coupling between

objects. For instance, **DataPointVO.java** heavily relies on **Common.java** for shared functionalities, and **PointValueTime.java** tightly interacts with **DataPointVO.java** for point value operations. In contrast, the loosely coupled classes demonstrate low coupling, with minimal dependencies on external components. For example, **UserDao.java**, **ViewDao.java**, and **WatchListDao.java** focus on specific functionalities like user management and view operations without extensive interactions with other classes. This distinction in coupling levels affects the maintainability and flexibility of the codebase, with tightly coupled classes being more prone to challenges in scalability and modification, while loosely coupled classes offer modularity and ease of maintenance.

Now based on the metric of RFC, we would like to evaluate the MangoDB Codebase:

1. **Three non-trivial classes with the tightest (highest = high RFC value) coupling:**

- *com.serotonin.mango.web.dwr.DataSourceEditDwr.java*
- *com.serotonin.mango.rt.RuntimeManager.java*
- *com.serotonin.mango.web.dwr.ViewDwr.java*

Explanation:

DataSourceEditDwr.java:

1. **Inheritance Coupling**:

- The class **DataSourceEditDwr** extends **DataSourceListDwr**, inheriting its methods and possibly its attributes (though attributes are not explicitly shown in the provided code snippet).
- The methods **toggleDataSource()** and **toggleDataPoint()** are called from the superclass (**DataSourceListDwr**), indicating a form of coupling where changes in the superclass could impact the subclass. This can be seen in the methods **toggleEditDataSource()** and **togglePoint()**.
- Inheritance coupling implies a form of tight coupling, as changes in the superclass can directly affect the subclass.

2. **Method Invocations**:

- Methods like **getPoints()**, **getAlarms()**, and **toggleDataSource()** make use of **Common.getUser().getEditDataSource()**. This shared method call indicates a form of coupling between these methods, creating a dependency.

- Similarly, methods like **tryDataSourceSave()** and **validatePoint()** make calls to other methods within the same class, leading to internal coupling.

3. **Dependency on External Classes**:

- There's dependency on other classes like **Common**, **User**, **DataPointVO**, **DataPointDao**, **EventInstance**, **EventDao**, etc., through method calls or usage of objects. Any changes in these external classes could potentially impact the behavior of **DataSourceEditDwr**.

4. **Interaction with Data Sources**:

- The class interacts with data sources through methods like **saveDataSource()** and **deleteDataPoint()**, which indicates a level of coupling with data storage mechanisms.

  In summary, the selected class (**DataSourceEditDwr**) exhibits high coupling due to:

- Inheritance coupling with **DataSourceListDwr**.
- Method invocations leading to dependencies on other classes and methods.
- Interactions with data sources, implying dependencies on external systems.

  This coupling is primarily of the type:

- **Inheritance Coupling**: Due to the inheritance relationship with **DataSourceListDwr**.
- **Content Coupling**: Through method invocations that rely on the internal implementation of other classes.
- **External Coupling**: Dependencies on external classes such as **Common**, **DataPointVO**, etc., which can be considered a form of external coupling.

RuntimeManager.java:

1. **Internal Coupling**:

- The class contains numerous private fields and methods (**runningDataSources**, **dataPoints**, **dataPointListeners**, etc.), and these methods interact with each other extensively within the class.

- Methods like **initialize()**, **startDataSourcePolling()**, **startPointLink()**, **startScheduledEvent()**, etc., interact with each other, indicating tight coupling within the class itself.

- The initialization process involves invoking several methods to set up various components of the runtime environment. Each of these methods interacts with internal state and other methods within the class.

2. **External Coupling**:

- The class interacts with external resources such as data sources, publishers, scheduled events, etc., through methods like **initializeDataSource()**, **startPublisher()**, **startScheduledEvent()**, etc.

- These interactions indicate coupling with external systems, where changes in these systems could potentially impact the behavior of **RuntimeManager**.

3. **Inheritance Coupling** (indirectly):

- While not explicitly shown in the provided code snippet, if other classes extend **RuntimeManager**, there would be a form of inheritance coupling. Changes in **RuntimeManager** could affect the behavior of its subclasses.

4. **Content Coupling**:

- Methods like **initialize()**, **startDataSourcePolling()**, etc., contain logic that depends on the internal implementation details of other classes (e.g., **DataSourceDao**, **PublisherDao**, etc.).

- This form of coupling suggests that changes in the implementation of these external classes could require modifications in **RuntimeManager**.

5. **Temporal Coupling**:

- The **initialize()** method performs a series of actions in a specific order. Changes in the order of method invocations or the logic within these methods could impact the behavior of the initialization process.

  In summary, the high RFC value of **RuntimeManager** indicates extensive coupling, both internally within the class and externally with other classes and systems.

ViewDwr.java:

1. **Dependency on Other Classes**:

- The class makes use of methods and fields from various other classes such as **Common**, **View**, **WebContextFactory**, **RuntimeManager**, etc.
- For example, methods like **Common.getAnonymousView()**, **view.findDataPoint()**, **RuntimeManager**, and **WebContextFactory.get()** are used extensively throughout the class.
- This dependency on external classes indicates a form of coupling where changes in these external classes could directly affect the behavior of **ViewDwr**.

2. **Method Invocations**:

- The class contains several methods (**getViewPointDataAnon()**, **setViewPointAnon()**, **getViewPointData()**, **getViewPointData()**) that interact with each other and with methods from external classes.
- Method invocations between these methods tangle them together, indicating a form of coupling where changes in one method could necessitate modifications in others.

3. **Data Passing**:

- Data passing between methods is evident, such as passing **view**, **user**, **edit** flags, etc., between different methods in the class.
- Passing data between methods tightly couples them together, as changes in the structure or requirements of the data could impact multiple methods.

4. **Content Coupling**:

- Methods like **addPointComponentState()** and **generateContent()** contain logic that relies on the internal implementation details of other classes.
- This form of coupling suggests that changes in the implementation of these external classes could require modifications in **ViewDwr**.

5. **Temporal Coupling**:

- The sequence and order of method invocations within **getViewPointData()** are crucial for the correct functioning of the class.
- Changes in the order or timing of these invocations could lead to unintended behavior, indicating a form of temporal coupling.

2. **Three non-trivial classes with the lowest (= high RFC value) coupling:**
    - *com.serotonin.mango.vo.report.DiscreteTimeSeries.java*
    - *com.serotonin.mango.db.MySQLAccess.java*
    - *com.serotonin.mango.util.ExportCodes.java*

    Explanation:

    DiscreteTimeSeries.java:

    1. **Class Structure**:

    - The class is self-contained and encapsulates a discrete time series data structure.
    - It has a simple constructor and a set of methods to manipulate and retrieve data from the time series.

    2. **Minimal Dependencies**:

    - The class depends only on a few basic Java classes (**String**, **Paint**, **ArrayList**) and a custom **PointValueTime** class, which is not shown in the provided snippet.
    - There are no external dependencies on complex or extensive libraries or frameworks.

    3. **Limited Method Invocations**:

    - The class has a limited number of methods, and they interact mainly with internal fields and collections (**valueTimes**, **valueDescriptions**).
    - There are no complex method invocations or interactions with external systems or classes.

    4. **Encapsulation**:

    - The class encapsulates its data fields (**name**, **textRenderer**, **paint**, **valueTimes**, **valueDescriptions**) and provides accessors to retrieve them, promoting encapsulation and reducing dependencies.

    5. **No Inheritance or Interface Implementation**:

    - The class does not extend any other class nor implement any interfaces, further reducing its dependencies and coupling with other classes.

    6. **No Complex Logic or Control Flow**:

- The class contains straightforward logic to add, retrieve, and manipulate data within the time series.
- There are no complex conditional statements or loops that could introduce dependencies on external factors.

MySQLAccess.java:

1. **Minimal External Dependencies**:

- The class depends only on the Java standard libraries and the **BasePooledAccess** class (which is not provided in the snippet but likely contains generic database access functionality).
- It does not rely on external libraries or frameworks specific to MySQL database access, reducing its external dependencies and coupling.

2. **Encapsulation of Database Access Logic**:

- The class encapsulates database access logic within itself without exposing it to external classes or systems.
- It provides methods like **initializeImpl()**, **getUrl()**, **getDriverClassName()**, etc., to handle database-specific configurations, reducing coupling with external configurations or systems.

3. **Limited Method Invocations**:

- The class contains a limited number of methods, and they interact mainly with the internal state and configurations of the MySQL database connection.
- There are no complex method invocations or interactions with external systems or classes, contributing to low coupling.

4. **No Complex Logic**:

- The class contains straightforward logic to initialize the MySQL database connection pool, configure database properties, and handle database-specific operations like checking for database existence and applying bounds to values.
- There are no complex conditional statements or loops that could introduce dependencies on external factors.

5. **Self-Containment**:

- The class is self-contained and does not extend or implement other classes or interfaces, reducing its dependencies and coupling with other classes.

ExportCodes.java:

1. **Minimal External Dependencies**:
    - The class relies solely on basic Java libraries (**java.util.List**, **java.util.ArrayList**, **org.apache.commons.lang3.ArrayUtils**) and does not depend on any external frameworks or libraries.
    - It does not interact with external systems, databases, or complex APIs, which reduces its external dependencies and coupling.
2. **Encapsulation of Functionality**:
    - The class encapsulates the functionality related to managing export codes within itself.
    - It provides methods like **addElement()**, **getCode()**, **getKey()**, **getId()**, etc., to add elements, retrieve codes, keys, and IDs, without exposing its internal implementation details to external classes or systems.
3. **Limited Method Invocations**:
    - The class contains a limited number of methods, and they interact mainly with the internal list of elements.
    - There are no complex method invocations or interactions with external systems or classes, contributing to low coupling.
4. **Self-Containment**:
    - The class is self-contained and does not extend or implement other classes or interfaces.
    - It encapsulates all functionality related to export codes management within itself, reducing its dependencies and coupling with other classes.
5. **No Complex Logic**:
    - The class contains straightforward logic to add elements, retrieve codes, keys, and IDs based on input parameters.

- There are no complex conditional statements or loops that could introduce dependencies on external factors.

6. **Data Structure Encapsulation**:

    - The class encapsulates elements as inner classes, which keeps related data together and avoids exposing them to external classes, promoting encapsulation and reducing coupling.

Measuring and interpreting Cohesion

OCmax :-

- OCmax calculates the cohesion based on the most interconnected pair of methods within the module.
- High OCmax indicates high cohesion because it means that there is a strong interdependence between methods, suggesting that they work closely together to achieve a common goal.
- Low OCmax suggests low cohesion as it implies that methods within the module are less interconnected and may have diverse or unrelated responsibilities.

OCavg :-

- OCavg calculates the cohesion based on the average interdependence among all pairs of methods within the module.
- High OCavg indicates high cohesion because, on average, the methods in the module are relatively interconnected, contributing to a cohesive unit of functionality.
- Low OCavg suggests low cohesion as it means that, on average, methods within the module have weaker interdependencies, which could be a sign of scattered or unrelated responsibilities.

jEdit Codebase:

Three non-trivial classes with the **highest cohesion using OCmax** : -

1. **ParserTokenManager.java** (org/gjt/sp/jedit/bsh/ParserTokenManager.java)
   For this class **OCmax is 241** which is the highest value in the project.
   The ParserTokenManager class is cohesive mostly in its functional sense. When components of a module (class) are connected through the performance of a particular

task or function, this is known as functional cohesiveness. In this instance, the class's sole purpose is token parsing, and every one of its variables and methods is directly tied to this ability. The class exhibits functional coherence in the following ways:
Related Methods: The parsing of tokens is closely related to all of the methods in the class (jjStopStringLiteralDfa_0, jjStartNfa_0, jjStopAtPos, jjStartNfaWithStates_0, jjMoveStringLiteralDfa0_0, etc.). Token parsing involves various stages and aspects, such as beginning, pausing, and progressing throughout the process, which are all aided by each approach.
Overall, by concentrating only on token parsing operations and grouping relevant functionality within the class, the ParserTokenManager class demonstrates functional coherence. Cohesion of this kind is advantageous since it increases the class's maintainability, readability, and reusability for token parsing tasks.

2. **TextUtilities.java** (org/gjt/sp/jedit/TextUtilities.java)
   For this class **OCmax is 50** which is the very value in the project
   This class demonstrates strong cohesiveness by assembling related techniques that work with comparable data or complete comparable tasks. The degree to which the duties of methods inside a class are focused and closely related is referred to as cohesion. In this instance, the course focuses on similar activities such as text formatting and string manipulation.
   This class demonstrates a **Functional Cohesion**. When methods within a class carry out comparable tasks or cooperate to accomplish a particular functionality or objective, this is known as functional cohesiveness. The following techniques provide this high level of functional cohesion:
   indexIgnoringWhitespace(String str, int index): This function returns the index of characters in a string that are not whitespace up to a specified index. It functions with indexing and character manipulation.
   By combining techniques that cooperate to carry out text formatting, character analysis, and string manipulation, the class exhibits strong Functional Cohesion overall.

3. **TextAreaInputHandler.java** (org/gjt/sp/jedit/input/TextAreaInputHandler.java)
   For this class **OCmax is 20** which is the very value in the project
   Both functional coherence and high cohesion are displayed by the TextAreaInputHandler class. The degree of emphasis and relatedness among a class's or module's obligations is referred to as cohesion. When methods within a class are grouped together based on completing a certain task or closely related duties, this is

referred to as functional cohesiveness.
Because all of the properties and methods in the TextAreaInputHandler class are concerned with managing input events for a text area (TextArea), there is a high degree of cohesiveness among them. Let's examine the ways in which this class exhibits functional cohesion:
The TextAreaInputHandler class exhibits functional cohesiveness by combining methods and fields that are directly linked to managing input events for the text area. It has strong cohesion since it concentrates on a single duty or task—managing important events for a text region.

Three non-trivial classes with the **lowest cohesion using OCmax** : -

1. **TextAreaDialog.java** (org/gjt/sp/jedit/gui/TextAreaDialog.java)
   The OCmax value for this class is 1 which is the lowest value of all.
   Because it has several tasks that are not directly related to one another, this TextAreaDialog class lacks cohesiveness. When a class has several unconnected functions or duties, it lacks coherence and the code becomes more difficult to comprehend, maintain, and reuse. In this instance, the class integrates dialog creation, action handling, error management, and UI logic—tasks that, in a perfect world, would be divided into various classes or modules to improve coherence.
   The different forms of cohesiveness this class demonstrated are broken down as follows: **Procedural Cohesion**:
   Methods like init(), ok(), and cancel(), where the stages are set to achieve a certain task (e.g., initializing UI components, closing the dialog on OK or cancel), are prime examples of procedural cohesiveness.
   By separating out certain processes inside their own scope, breaking these methods down into smaller, more focused ways can help with readability and maintainability.

2. **Abbrevs.java** (org/gjt/sp/jedit/Abbrevs.java)
   Due to the fact that this Abbrevs class contains multiple unrelated functionality within it, it lacks cohesiveness. The degree of relationship between a class's data and methods is called cohesion. Low cohesiveness indicates that the class is working on a variety of unrelated, diversified tasks, which makes it difficult to focus and clearly define its design.
   Different types of cohesiveness can be identified in the given class:
   Communicational Cohesion: To provide error feedback and allow user involvement (AddAbbrevDialog), the class communicates with UI components

(javax.swing.UIManager).
For text expansion and manipulation, it additionally communicates with the buffer and text area components (Buffer, JEditTextArea, and View).
These communication duties, however, have little bearing on one another or the main objective of abbreviation management.

3. **StatusBar.java** (org/gjt/sp/jedit/gui/StatusBar.java)
   The main reason this StatusBar class lacks cohesiveness is that it contains several unrelated functionalities in one class. When duties and assignments in a class are not tightly related to one another or are not focused, it is referred to as having low cohesiveness. This can cause confusion and make it more difficult to maintain and comprehend the code. The class is exhibiting both temporal and functional cohesiveness in this instance.
   When various techniques or components of a class are grouped together because they carry out similar duties, this is known as functional cohesion. Nevertheless, it appears that the propertiesChanged() and addNotify() methods in the provided class handle distinct tasks unrelated to one another or the primary goal of the class.

Three non-trivial classes with the **highest cohesion using OCavg**: -

1. **ParserTokenManager.java** (org/gjt/sp/jedit/bsh/ParserTokenManager.java)
   For this class **OCmax is 241** which is the highest value in the project.
   The ParserTokenManager class is cohesive mostly in its functional sense. When components of a module (class) are connected through the performance of a particular task or function, this is known as functional cohesiveness. In this instance, the class's sole purpose is token parsing, and every one of its variables and methods is directly tied to this ability. The class exhibits functional coherence.
   By concentrating only on token parsing operations and grouping relevant functionality within the class, the ParserTokenManager class demonstrates functional coherence. Cohesion of this kind is advantageous since it increases the class's maintainability, readability, and reusability for token parsing tasks.
2. **MouseWheelHandler.java** (org/gjt/sp/jedit/textarea/TextArea.java)
   The value for Ocavg here is 13 which is very high compared to other classes.
   This MouseWheelHandler class merges several unrelated functionality into a single method, mouseWheelMoved, which results in high cohesiveness. A class or approach that attempts to handle too many unrelated tasks is said to have high coherence. The

mouseWheelMoved method in this instance manages a variety of mouse wheel events, however it combines disparate behaviors, which leads to a High Cohesion.
The type of cohesion exhibited here is control cohesion or coincidental cohesion. Control cohesion occurs when elements within a module are combined based on the need for common control logic, such as switch statements or if-else conditions. In this case, the method combines different functionalities related to mouse wheel handling, scrolling, caret movement, and text selection, resulting in control-driven cohesion.

3. **ActionHandler.java** (org/gjt/sp/jedit/gui/CloseDialog.java)
   This class ActionHandler exhibits High cohesion because it contains multiple unrelated functionalities within the actionPerformed method. Cohesion refers to the degree to which the elements of a module (in this case, a class) are related and work together to achieve a single purpose. High cohesion means that the elements of the class are not strongly related or do not work towards a single goal efficiently.
   In the provided ActionHandler class, the actionPerformed method handles four distinct actions based on different sources (selectAll, save, discard, and cancel). These actions seem to be related to some user interface components or application functionalities, but they are not directly related to each other in terms of functionality or purpose.
   The type of cohesion exhibited in this class is known as "Coincidental Cohesion". Coincidental cohesion occurs when parts of a module are grouped together arbitrarily, often because they happen to be implemented in the same area of code or because they are handled by the same method, even though they are not logically related.

Three non-trivial classes with the **Lowest cohesion using OCavg**: -

1. **TextAreaDialog.java** (org/gjt/sp/jedit/gui/TextAreaDialog.java)
   The Ocavg value for given class is 1.00 which is the lowest value for Ocavg.
   This TextAreaDialog class exhibits low cohesion because it combines multiple unrelated functionalities within a single class. Cohesion refers to the degree to which the elements within a module (class in this case) are related or focused on a single purpose. Low cohesion means that the class is handling multiple responsibilities or concerns that are not strongly related to each other.
2. **WarpWidgetFactory.java**
   (org/gjt/sp/jedit/gui/statusbar/WrapWidgetFactory.java)
   The Ocavg value for given class is 1.00 which is the lowest value for Ocavg.
   This Java class WrapWidgetFactory exhibits low cohesion due to its inclusion of multiple responsibilities within a single class.

Font Metrics and Dimension Calculations:
The WrapWidget class also includes logic for calculating font metrics and dimension adjustments based on the current font. This additional responsibility of managing UI dimensions adds to the class's lack of cohesion.
The type of cohesion exhibited in this class is Coincidental Cohesion. Coincidental cohesion occurs when different responsibilities are grouped together in a class without a clear logical connection. In this case, the class combines unrelated functionalities such as widget creation, event handling, UI updates, property management, and dimension calculations.

3. **TaskMonitor.java** (org/gjt/sp/jedit/gui/TaskMonitor.java)
   The Ocavg value for given class is 1.08 which is the lower compared to other values for Ocavg in the whole project.
   The TaskMonitor class exhibits low cohesion because it combines multiple responsibilities that are not directly related to each other. Low cohesion refers to a design issue where a class has too many responsibilities or functionalities that are not closely related. In this case, the class is responsible for managing a GUI component (JPanel), handling tasks (TaskListener methods), and managing the data displayed in a table (TaskTableModel).
   the class exhibits low cohesion due to mixing GUI component management, task handling, data management, and event handling in a single class. To improve cohesion, the responsibilities should be divided into separate classes, each handling a specific concern such as GUI management, task management, data management, and event handling.

MangoDB Codebase:

Three non-trivial classes with the **highest cohesion using OCmax** : -

1. **MiscDwr.java** (com/serotonin/mango/web/dwr/MiscDwr.java)

OCmax value for this class is 28 which is highest in the non trivial classes.

This Java class MiscDwr exhibits high cohesion by grouping related functionalities together, which is a characteristic of functional cohesion. Let's break down how this class demonstrates functional cohesion and why it doesn't exhibit other types of cohesion:

Functional Cohesion:

- The class primarily deals with various operations related to handling events, user actions, and documentation items. These functionalities are closely related and form a logical unit within the context of the application's functionality.

- Methods like toggleSilence, silenceAll, acknowledgeEvent, acknowledgeAllPendingEvents, toggleUserMuted, and getDocumentationItem all revolve around managing different aspects of event handling, user interactions, and documentation retrieval.

- The class also includes a method jsError that handles JavaScript errors, which is somewhat related to the overall functionality of the class in terms of error handling.

Contrast with Other Types of Cohesion:

- Coincidental Cohesion: This class does not exhibit coincidental cohesion because its methods are not randomly grouped together. They are all related to managing different aspects of events, user actions, and documentation items, which align with the application's purpose.

- Logical Cohesion: While the methods are logically grouped to handle related tasks, the class does more than just processing data in a sequential or logical manner. It encompasses a range of functionalities related to event management, user interactions, and error handling.

- Temporal Cohesion: There is no specific temporal relationship or dependency among the methods in terms of timing or sequence of execution.

- Procedural Cohesion: This class does not exhibit procedural cohesion, as it doesn't represent a single step in a larger process or procedure.

- Communicational Cohesion: Although there is some communication happening within methods (e.g., jsError method logs a warning), the primary focus of the class is not on communication between different modules or components. It's more about performing specific actions based on user input or system events.

- Informational Cohesion: While the getDocumentationItem method deals with retrieving information, the class as a whole is not primarily focused on processing or managing information.

2. **VMStatDataSourceRT.java**

(com/serotonin/mango/rt/dataSource/vmstat/VMStatDataSourceRT.java)

The OCmax value for given class is 24.

This Java class, VMStatDataSourceRT, exhibits high cohesion by encapsulating related functionality within the class itself. Cohesion refers to the degree to which the elements within a module or class belong together. In this case, the class demonstrates functional cohesion, where all methods and attributes are related to the functionality of managing a VMStat data source.

Here's how the class exhibits high functional cohesion:

- Purpose: The class is designed specifically to handle VMStat data sources, including initializing, polling, and handling data.
- Attributes: The class contains attributes such as vo (VMStatDataSourceVO), vmstatProcess, in, attributePositions, terminated, and log, all of which are directly related to managing VMStat data and its processing.
- Methods: Methods like initialize(), terminate(), beginPolling(), run(), readParts(), and readError() are focused on specific tasks related to handling VMStat data sources, such as initializing the data source, terminating it, starting polling, running the data processing loop, parsing data parts, and handling errors.

Now, let's contrast why it is not exhibiting other types of cohesion:

- Coincidental Cohesion: This type of cohesion occurs when elements in a module or class are grouped together arbitrarily, without a clear logical relationship. The VMStatDataSourceRT class does not exhibit coincidental cohesion because all its methods and attributes are directly related to managing VMStat data sources; there are no unrelated elements grouped together arbitrarily.
- Logical Cohesion: Logical cohesion involves grouping elements that are logically related but not functionally related. The class does not exhibit logical cohesion because all its elements are functionally related to handling VMStat data sources.
- Temporal Cohesion: Temporal cohesion occurs when elements are grouped together because they are executed at the same time. While the class has methods like initialize() and run() that are executed at different times, these methods are still functionally related to managing VMStat data sources, not just grouped together based on temporal execution.

3. **DataSourceEditDwr.java**

(com/serotonin/mango/web/dwr/DataSourceEditDwr.java)

The DataSourceEditDwr class exhibits high cohesion, specifically functional cohesion, based on its implementation of related functionality within the same class. Functional cohesion refers to the degree to which elements within a module (in this case, a class) are functionally related and contribute to a single well-defined task or purpose.

Here's how the class demonstrates functional cohesion:

editInit() Method: This method initializes the editing process for a data source and returns an internationalized response object containing data relevant to the editing operation.

tryDataSourceSave() Method: This method attempts to save a data source, validates it, and adds relevant data to the response object if the save operation is successful.

Contrasting with other types of cohesion:

- Coincidental Cohesion: This occurs when elements within a module are grouped together arbitrarily and do not contribute to a single purpose.
- Logical Cohesion: This type of cohesion involves grouping elements based on their logical relatedness, which may not necessarily correspond to a single well-defined task.

Three non-trivial classes with the **Lowest cohesion using OCmax**

1. **PointValueDao.java** (com/serotonin/mango/db/dao/PointValueDao.java)

The OCmax value for this class is 1 which is the lowest value in the whole project.

The given PointValueDao class exhibits functional cohesion, which is the lowest level of cohesion. Functional cohesion occurs when elements of a module (in this case, methods within the PointValueDao class) are grouped because they all contribute to a single well-defined task or function, which in this case is saving point values.

Here's how the class demonstrates functional cohesion and why it does not exhibit other types of cohesion:

Functional Cohesion:

All methods in the class are related to saving point values in different scenarios (synchronously, asynchronously, handling image data, etc.).

The class has methods like savePointValueSync and savePointValueAsync that directly contribute to the task of saving point values.

The savePointValueImpl method is responsible for the actual implementation of saving point values, including handling different data types and sources.

Contrast with Other Cohesion Types:

- Coincidental Cohesion: There are no methods or elements in the class that are coincidentally grouped. Every method serves a clear purpose related to saving point values.
- Logical Cohesion: While there is some logical flow in the class, such as handling retries and concurrency, these are still part of the overall task of saving point values rather than being a separate logical unit.

2. **DataSourceDao.java** (com/serotonin/mango/db/dao/DataSourceDao.java)

The OCmax value for this class is 2.

This Java class, DataSourceDao, exhibits low cohesion, specifically functional cohesion. Functional cohesion is achieved when elements within a module (in this case, the class) perform tasks that are closely related and contribute to a single well-defined purpose, such as managing data sources in this case. It shows the functional cohesion.

Contrast with Other Types of Cohesion:

- Coincidental Cohesion: This class does not exhibit coincidental cohesion, which occurs when elements within a module are grouped together arbitrarily and do not have a clear, meaningful relationship. In this class, all methods and attributes are related to data source management, indicating a purposeful grouping of elements.
- Logical Cohesion: While the class does involve logical operations such as condition checks and data manipulation, these operations are all related to data source management, making it primarily functionally cohesive rather than logically cohesive.
- Temporal Cohesion: Temporal cohesion involves grouping elements based on when they are executed. In this class, methods related to data source operations are grouped together regardless of when they are executed, so it does not exhibit temporal cohesion.

3. **CustomViewPoint.java**

(com/serotonin/mango/view/custom/CustomViewPoint.java)

The provided Java class CustomViewPoint exhibits low cohesion, specifically functional cohesion. Functional cohesion is characterized by grouping related functionalities and operations together within a class. The class demonstrates low functional cohesion.

Contrast with other types of cohesion:

- Logical Cohesion: This class doesn't exhibit logical cohesion, which involves grouping functionalities based on logical relationships that may not be related by purpose. The functionalities in CustomViewPoint are logically related to creating custom views for data points.

- Temporal Cohesion: Temporal cohesion involves grouping functionalities based on the timing of their execution. The CustomViewPoint class does not exhibit temporal cohesion as it focuses on creating custom views regardless of timing or sequence of operations.

- Procedural Cohesion: Procedural cohesion involves grouping functionalities based on procedural flow or sequence. While the CustomViewPoint class follows a procedural flow in its createStateImpl method, its primary focus is on functional aspects related to custom view creation.

Three non-trivial classes with the **highest cohesion using OCavg** : -

1. **ImageCharServlet.java** (com/serotonin/mango/web/servlet/ImageChartServlet.java)

The OCavg value for this class is 7.33 which is a very high value.

This ImageChartServlet class exhibits high functional cohesion. Functional cohesion refers to a situation where elements of a module are related by performing a single function or task. In this class, all the methods and fields are related to the functionality of generating and caching image charts based on incoming requests.

Contrast with Other Types of Cohesion:

- Logical Cohesion: The class does exhibit some logical cohesion as it deals with tasks related to image chart generation and caching. However, it focuses more on functional aspects (generating, caching, and purging) rather than logical relationships between elements.

- Temporal Cohesion: While there is a temporal aspect in the tryCachePurge method (purging cache based on time intervals), it's not the primary focus of the class. The main focus is on image chart generation and caching.

- Procedural Cohesion: The class does not exhibit procedural cohesion as it's not primarily focused on a sequence of steps or procedures. Instead, it focuses on a specific task related to image chart handling.

- Communicational Cohesion: This type of cohesion is about elements that operate on the same input data or share intermediate results. In this class, methods communicate to achieve the overall task of generating and caching image charts, but it's not the primary mode of cohesion.

2. **DataSourceUtils.java**

(com/serotonin/mango/rt/dataSource/DataSourceUtils.java)

The Ocavg value for this class is 7.00

This Java class DataSourceUtils exhibits high cohesion by encapsulating related functionalities within its methods. Cohesion refers to the degree to which elements within a module (such as a class) belong together and work towards a common purpose. Let's analyze how this class demonstrates high cohesion and what type of cohesion it represents:

Functional Cohesion:

The class focuses on providing utility methods related to data source handling (getValue, getValueTime).

Each method performs a specific, well-defined function related to data parsing and value retrieval.

The methods work together to achieve the common goal of processing data and returning appropriate MangoValue objects.

This class is not exhibiting other types of cohesion such as:

- Coincidental Cohesion: There are no unrelated functionalities grouped together just by coincidence. All methods are directly related to data source handling and value extraction.

- Logical Cohesion: While the class does follow logical patterns and flows for data processing, its main focus is on functional and informational cohesion rather than organizing code based solely on logical operations.

- Temporal Cohesion: There is no specific time-related grouping or operations that would classify this class as exhibiting temporal cohesion. The methods operate independently of time-related considerations.

3. **EventManager.java** (com/serotonin/mango/rt/EventManager.java)

This EventManager class exhibits high cohesion primarily through functional cohesion. Let's break down the aspects of cohesion observed and why it contrasts with other types of cohesion:

Functional Cohesion (High Cohesion):

The class focuses on a single well-defined task: event management, including raising events, returning events to normal, canceling events, and handling event lifecycle.

Methods such as raiseEvent, returnToNormal, cancelEventsForDataPoint, cancelEventsForDataSource, and cancelEventsForPublisher are all related to event management and manipulation.

Contrast with Other Cohesion Types:

- Coincidental Cohesion: There are no randomly grouped methods or attributes that lack a clear relationship. Everything revolves around event management.
- Logical Cohesion: While there is logical flow within methods, the class's primary cohesion is functional, centered on event management rather than logical operations.
- Temporal Cohesion: There are no methods grouped solely by the timing of their execution. All methods are related to event handling throughout the application's lifecycle.
- Procedural Cohesion: Although methods follow a sequence of steps, they are tightly related to event handling rather than general procedural tasks.
- Communicational Cohesion: While there is communication with DAOs and user-related operations, these are part of event management and don't form a separate cohesive group.

Three non-trivial classes with the **Lowest cohesion using OCavg** : -

1. **ReportDao.java** (com/serotonin/mango/db/dao/ReportDao.java)

The provided class ReportDao exhibits low cohesion because it combines several unrelated responsibilities and functionalities within a single class. Cohesion refers to how closely related and focused the responsibilities of a class are. Low cohesion occurs when a class has multiple responsibilities that are not closely related to each other.

Here are the observations on the cohesion type and why it is not another type out of the listed types:

Low Cohesion Type: Coincidental Cohesion

Coincidental cohesion occurs when different parts of a class are grouped together without any meaningful relationship.

Contrast with Other Cohesion Types:

- Logical Cohesion: Logical cohesion involves grouping methods that logically belong together. The PointInfo class does not exhibit logical cohesion because its methods do not logically relate to each other.
- Temporal Cohesion: Temporal cohesion involves grouping methods that are executed at the same time. The methods in PointInfo are not necessarily executed at the same time, so it does not exhibit temporal cohesion.
- Procedural Cohesion: Procedural cohesion involves grouping methods that are related to a specific procedure or process. The methods in PointInfo do not form a cohesive procedure, so it does not exhibit procedural cohesion.
- Communicational Cohesion: Communicational cohesion involves grouping methods that operate on the same data. While PointInfo deals with data related to a point, it also includes unrelated information like color and chart consolidation, making it not purely communicational.

2. **UserDao.java** (com/serotonin/mango/db/dao/UserDao.java)

The OCavg value for the given class is 1.19 which is very low.

The provided Java class UserDao exhibits low cohesion due to the presence of multiple responsibilities and functionalities within the same class. Cohesion refers to how closely related and focused the responsibilities of a class are. Low cohesion occurs when a class handles unrelated tasks or performs multiple functionalities that are not closely related. Let's analyze the cohesion type observed in this class and contrast it with other cohesion types to understand why it is not exhibiting them.

It exhibits functional cohesion.

Contrast with Other Cohesion Types:

- Coincidental Cohesion: This class does not exhibit coincidental cohesion because the methods and functionalities present are not randomly or coincidentally grouped. They are related to user management, albeit in a loosely coupled manner.

- Logical Cohesion: The class does not demonstrate logical cohesion as it does not solely focus on a specific logical aspect or operation. Instead, it covers a range of functionalities related to user management.

- Temporal Cohesion: Temporal cohesion is not observed because the methods in the class are not necessarily executed in a specific sequence or temporal order.

3. **ShareUser.java** (com/serotonin/mango/view/ShareUser.java)

The OCavg value for given class is 1.67 which is very less compared to the other classes.

This Java class ShareUser exhibits low cohesion because it combines several unrelated functionalities within the same class. Cohesion refers to the degree to which elements of a module (such as a class) are related and work together towards a common purpose. Low cohesion means that the class contains disparate or unrelated functionalities.

To contrast, let's analyze why this class is not exhibiting other types of cohesion:

- Coincidental Cohesion: Coincidental cohesion occurs when elements of a module are grouped together arbitrarily without any meaningful relationship. In this class, although there are disparate functionalities, they are related to the overall purpose of managing shared user access, making it more than coincidental cohesion.

- Logical Cohesion: Logical cohesion involves grouping elements based on their logical relationship and functionality. While the class does have some logical grouping (e.g., user-related fields and access control constants), it still combines unrelated functionalities, indicating low cohesion rather than logical cohesion.

- Temporal Cohesion: Temporal cohesion involves grouping elements that are used at the same time. This class does not exhibit temporal cohesion as the elements are not necessarily used together at the same time; for instance, database access and JSON serialization may not occur simultaneously.

- Procedural Cohesion: Procedural cohesion occurs when elements are grouped based on the order in which they are executed. The class does not exhibit procedural cohesion as it combines functionalities that may not follow a strict procedural order.